2015-03

Toward a robust method of presenting a rich, interconnected deceptive network topology

West, Austin

Monterey, California: Naval Postgraduate School

http://hdl.handle.net/10945/45271

NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

TOWARD A ROBUST METHOD OF PRESENTING A RICH, INTERCONNECTED DECEPTIVE NETWORK TOPOLOGY

by

Austin West

March 2015

Thesis Advisor:	Robert Beverly
Second Reader:	Geoffrey Xie

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE 03-27-2015	3. REPORT TYPE AND DATES COVERED Master's Thesis 11-01-2013 to 03-15-2015

4. TITLE AND SUBTITLE

TOWARD A ROBUST METHOD OF PRESENTING A RICH, INTERCONNECTED DECEPTIVE NETWORK TOPOLOGY

5. FUNDING NUMBERS

N66001-2250-58231

6. AUTHOR(S)

Austin West

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of Homeland Security
245 Murray Lane SW, Washington, DC 20528

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Every day, adversaries bombard Department of Defense computer networks with scanning traffic in order to gather information about the target network. This reconnaissance is typically a precursor to attacks designed to access data, exfiltrate information, or plant malware in order to gain a military advantage. One specific reconnaissance tool, traceroute, is used to map the network topology of a target network. We implement an active network defense tool, dubbed DeTracer, that seeks to thwart network mapping attacks through the use of deception. We deploy DeTracer in several environments, including the Internet, to demonstrate that an attacker attempting to map a target network using traceroute probes can be presented with a false network topology of the defender's choosing. Our experiments show that a defender can present an adversary with a credible false network topology. We are able to deceive all types of incoming traceroute probes, present a complex false network topology on a per source and destination basis, and deploy our deception scheme without disrupting service to the real production infrastructure on our network.

14. SUBJECT TERMS

Topological deception, active defense, traceroute, network defense

15. NUMBER OF PAGES 103

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU

THIS PAGE INTENTIONALLY LEFT BLANK

TOWARD A ROBUST METHOD OF PRESENTING A RICH, INTERCONNECTED DECEPTIVE NETWORK TOPOLOGY

Austin West
Civilian, Department of Defense
B.S., University of California at San Diego, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2015

Author: Austin West

Approved by: Robert Beverly
Thesis Advisor

Geoffrey Xie
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Every day, adversaries bombard Department of Defense computer networks with scanning traffic in order to gather information about the target network. This reconnaissance is typically a precursor to attacks designed to access data, exfiltrate information, or plant malware in order to gain a military advantage. One specific reconnaissance tool, traceroute, is used to map the network topology of a target network. We implement an active network defense tool, dubbed DeTracer, that seeks to thwart network mapping attacks through the use of deception. We deploy DeTracer in several environments, including the Internet, to demonstrate that an attacker attempting to map a target network using traceroute probes can be presented with a false network topology of the defender's choosing. Our experiments show that a defender can present an adversary with a credible false network topology. We are able to deceive all types of incoming traceroute probes, present a complex false network topology on a per source and destination basis, and deploy our deception scheme without disrupting service to the real production infrastructure on our network.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AS	autonomous system
BGP	Border Gateway Protocol
CIDR	Classless Inter-Domain Routing
CJCS	Chairmain, Joint Chiefs of Staff
DHS	Department of Homeland Security
DNS	Domain Name System
DOD	Department of Defense
DOS	Denial of Service
EW	electronic warfare
FTP	File Transfer Protocol
HTTP	Hypertext Transfer Protocol
IANA	Internet Assigned Numbers Authority
iBGP	internal Border Gateway Protocol
ICMP	Internet Control Message Protocol
ISP	Internet Service Provider
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
JP	Joint Publication
KB	kilobytes

MB	megabytes
Mbps	megabits per second
MILDEC	military deception
ms	milliseconds
NIC	network interface card
NPS	Naval Postgraduate School
NSS	National Security Strategy
OS	operating system
OSI	Open Systems Interconnection
RFC	Request For Comment
RIP	Routing Information Protocol
RTT	Round Trip Time
SDN	Software-Defined Networking
SNOS	Systemic Network Obfuscation System
SSH	Secure Shell
TCP	Transmission Control Protocol
TTL	time to live
TTP	tactics, techniques, and procedures
UDP	User Datagram Protocol
VM	virtual machine
VPN	Virtual Private Network
VRF	virtual routing and forwarding

Acknowledgments

I would like to thank Dr. Robert Beverly for his constant stream of support and wisdom. I would also like to thank Dr. Geoffrey Xie for providing a secondary perspective on this research and for starting me on the track towards a networking-themed thesis. Their insights were invaluable to my research and writing efforts.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: INTRODUCTION

This chapter discusses the prevalence of computer network probing, introduces the concepts of network topology mapping and active network defense, and examines their application to the Department of Defense (DOD).

1.1 Computer Network Probing

According to Internet Live Stats, approximately 360 million people worldwide were connected to the Internet in the year 2000 [1]. Just fifteen years later, that number is almost three billion, as around 40% of the world population has an Internet connection [1]. Not only has the sheer number of Internet users grown exponentially, the importance of the applications and data that we entrust to it, as well as our reliance upon it, has increased dramatically. We now comport much of our daily lives in the cyber domain, from social networking to managing bank accounts. Even the management of our critical infrastructure such as energy, transportation, and finance is wholly dependent upon technologies built in cyberspace.

While all of these technological advancements have vastly improved the daily lives of people across the world, they have exposed many of the things we rely on to avenues of attack that were inconceivable even ten or twenty years ago. Attackers are able to break into networks, break functionality, and exfiltrate information from anywhere in the world. Additionally, the cyber attack is becoming the weapon of choice for nation-states as the modern battlefield evolves to include cyber space. U.S. public and private networks "are probed millions of times every day," and "an amount of intellectual property larger than that contained in the Library of Congress is stolen from networks maintained by U.S. businesses, universities, and government departments and agencies" [2].

Computer network scanning is a procedure in which an attacker identifies the active hosts on a network. The goal of this intelligence-gathering activity is to discover open ports, vulnerable services, and the operating system (OS) of the computers on a network. Scanning can be done either passively, (without sending packets to the target network but by eavesdropping traffic), or actively (an attacker sends probes to the target network and gleans information from the responses). Scanning is generally a precursor to further attacks.

One specific type of computer network attack is a topology mapping attack, which is often a part of an attacker's intelligence-gathering phase and a precursor to further exploitation. A network topology is its layout and illustrates how it is designed and interconnected. In an active topology mapping attack, an attacker sends network probes to the target with the hopes of eliciting responses that reveal information about the structure and characteristics of the target network. The most common tool for this attack is traceroute, which sends active probes to a network with the goal of mapping network topologies. Armed with this information, an attacker may better direct his future attacks at poorly defended or critical nodes in the network. For example, an attacker may cause a denial of service to an entire network, partition a network to prevent connectivity between two subnetworks, and generally wreak havoc upon the target network [3].

Well-defended networks deny these scanning probes as well as topology mapping probes from entering their network in order to prevent any intrusions. However, it is difficult to determine what traffic is legitimate and to implement an effective security posture without adversely affecting the legitimate traffic that occurs normally on the network. Attackers commonly disguise attack traffic to appear like normal traffic in order to avoid detection. This means that denying some traffic that appears to be attack traffic could also block legitimate traffic and frustrate end users of the network. Additionally, network defenders can never know if they are successfully denying attackers access to their networks.

1.2 Active Network Defense

Another innovation in cyber security is the development of active network defenses. Since traditional antivirus tools lag behind the cutting edge of computer attacks, and firewalls are inherently unable to block all malicious traffic from entering a network, researchers have attempted to take a more aggressive stance on network defense.

> We can broadly define [active network defense] as any measures originated by the defender against the attacker. Because the purpose of any computer network defense is to protect information systems, these active measures must

> at least thwart any attack in progress, and ideally make further attacks more difficult. We can divide them into three broad categories: counterattack, preemptive attack, and active deception. [4]

In this thesis, we will focus on the third category of active network defense, specifically active deception as it relates to active network topology mapping. We will attempt to bolster computer network defenses by presenting false network topologies to attackers performing network mapping attacks.

1.3 Network Topology Deception

We posit that a network defense strategy rooted in deception can provide benefits to network defenders on top of those gained by implementing secure, patched computer systems. Trassare [3], demonstrates that it is possible to deploy a network defense tool that deceives an attacker's traceroute probes. While his implementation of a deceptive router is a simple proof of concept, his work outlines the benefits of presenting a false network topology [3]. In this thesis, we expound upon the advantages of deploying topology deception and drastically improve the implementation of the deceptive router in Trassare's work [3]. Our methodology does not necessarily add security for end hosts on the network; it does protect the core infrastructure of our networks, like routers and links, by disguising their layout. An attacker with a false view of a target network's topology might be induced into directing his attack in a way that the defender chooses: either toward resilient nodes and links, non-existent nodes and links, or low-value targets. We present the attacker with a faked topology by manipulating the responses to network mapping probes in a manner that will lead an attacker to draw incorrect conclusions about the layout of the network. This fake topology is configurable and easy to change if necessary. We believe the topology deception can keep an attacker entrenched in the intelligence-gathering phase of his attack for longer as well as redirect his attack in a way that is advantageous to the defender.

Our topology deception tool, hereafter referred to as DeTracer, is implemented on a smart router by detecting traceroute probes and crafting fake responses to them. In this way, DeTracer can present a fake network that can be configured to appear essentially however the defender chooses. In our experimentation, DeTracer was deployed on a production network on the Internet but never tested for resiliency or against human subjects. Further work is needed in this area to determine the effectiveness of the deception and the robustness of the DeTracer program itself.

1.4 Application to the DOD

We believe that network topology deception research is highly applicable to the DOD's core mission. As of 2011, the DOD "operates over 15,000 networks and seven million computing devices across hundreds of installations in dozens of countries around the globe" [2]. With such a broad surface area for attack in cyber space, it is essential that we remain on the cutting edge of network defense tactics and technology. This thesis discusses DeTracer, an innovative tool that is a step toward applying deception in cyber space to defend against network mapping attacks.

1.5 Summary of Contributions

This thesis:

- Introduces a novel methodology for detecting traceroute probes that is seamlessly integrated into DeTracer.
- Implements the topology deception scheme outlined by Chapter 3 in a Python program.
- Examines the problem of introducing artificial delay into traceroute probe responses in order to maximize a deception scheme's plausibility.
- Allows for the topology deception to be deployed at multiple ingress points to a network and present a consistent, fake map of the network.

The remainder of this thesis is organized as follows. Chapter 2 delves into the background of network topology mapping and how it applies to implementing a realistic deception tool. Chapter 3 discusses the methodology behind building DeTracer. In Chapter 4, we lay out the findings of our research, and Chapter 5 contains our conclusions.

CHAPTER 2: BACKGROUND

The benefits of employing deception in the cyber domain have been explored in the past. There are two main categories of deception strategies for cyber defense. The first technique is the use of obfuscation to hide vulnerable operating systems, services, or hosts on a network. This method includes measures such as changing public banners that leak information to something "non-committal," removing content from web pages that might reveal information about the operating system or web server, and "insulating the host from probe packets via a firewall or packet 'Scrubber"' [5]. A packet Scrubber is a tool that will rewrite packets in order to prevent characteristics of the packets a host generates from revealing information about that host [5]. The second method of deception is the use of honeypots. A honeypot is a system deployed by a defender that appears enticing, i.e., vulnerable, to a potential attacker [6]. However, it is simply a program running on a host that is dedicated to redirect attackers away from a more fruitful avenue of attack, slow down their progress attacking a network, or to investigate an attacker's tactics, techniques, and procedures (TTP) [6].

In this paper, we focus on one particular type of deception: fake network topology. While there has been some work in developing a tool that will present a deceptive network topology, the idea is still in its infancy and does not exist beyond a simple proof of concept implementation [3]. In the work by Trassare [3], they demonstrated that it is possible to deploy a smart router capable of deceiving incoming traceroute probes. However, their implementation contained several shortcomings. They were only able to deceive User Datagram Protocol (UDP) traceroute probes, did not alter the false traceroute results based on the source and destination of the traceroute probe, and did not return realistic Round Trip Time (RTT) values to traceroute probes. In our deception scheme, we solve these weaknesses and make the contributions outlined in Section 1.5. We believe that presenting a fake network topology to attackers allows us to intentionally mislead the attacker, keep them in an intelligence-gathering phase, and potentially thwart future attacks with deception. If this methodology does prove viable it will be a valuable addition to any network administrator's defense efforts. This chapter examines some of the tools employed by attackers and defenders as well as investigate some of the benefits that deception can introduce.

2.1 Topology Basics

Network topology is the physical or logical layout of nodes in a computer network. There are several levels of abstraction at which one can view a network's topology. The first is the physical topology, which shows the physical layout of the network, including how it is constructed.

Another, more interesting way to view a network is logically. A logical network diagram is essentially how the nodes in the network are connected and function [7]. Figure 2.1 depicts an example.

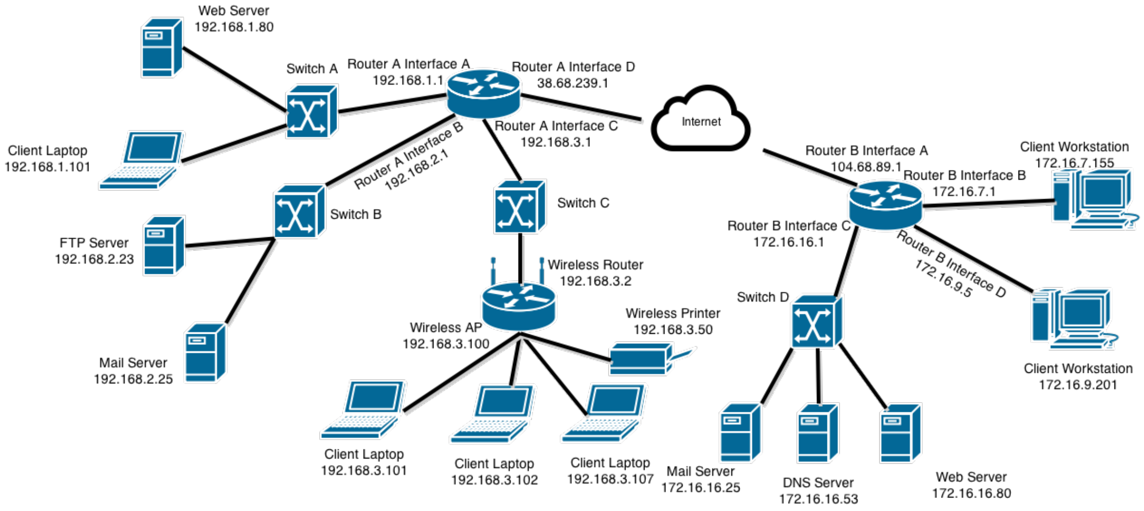


Figure 2.1: An example of a logical network diagram.

It is important to note that there are two types of connectivity, which are referred to as Layer 2 and Layer 3 connections, and correspond to the Data Link and Network Layer of the Open Systems Interconnection (OSI) model. The OSI model is an abstraction that partitions the roles of a communication system into layers. The Data Link Layer connects two directly connected nodes, while the Network Layer connects two devices that are on the same network, which are not necessarily directed. The Network Layer is responsible for the next-hop routing of packets through a network that allows a packet to traverse a network from the source to the destination [8]. Generally, switches operate as Layer 2 devices and routers are Layer 3 devices. Since traceroute probes are done using features of the Layer 3 packet, only nodes that operate at Layer 3 or higher in the networking stack appear in traceroute results. As such, the switches in Figure 2.1 would not appear in any traceroute probe. While they allow for connectivity between machines, they do not have Internet Protocol (IP) addresses.

Adding to the complexity of network topology is that many devices have multiple interfaces. This means that a single device (usually routers) are connected to multiple networks (subnets) at the same time in order to act as gateways between them.

Network researchers and cyber attackers have a keen interest in mapping network traffic and how devices on the Internet are connected. This is important for several reasons. For example, a corporation might be interested in the path that a packet takes from a source to a destination so that they may optimize the network performance for their customers. A researcher might be interested in how the path adjusts in response to congestion in certain parts of the network and how alternate paths could be optimized. An attacker might be interested in the path a packet takes because they may be able to insert a tap somewhere along that path in order to eavesdrop traffic. They also might be able to discover single points of failure that would disrupt service for their targets if certain nodes were taken down. Further, the topology of a particular organization's network can be of value to attackers interested in compromising one machine within the organization's network in order to use that node as a pivot point to attack other devices or to simply eavesdrop on the traffic that flows through a network.

If we view the network from a broader perspective, we introduce the concept of an autonomous system (AS). An AS is "a connected group of one or more IP prefixes run by one or more network operators which has a single and clearly defined routing policy" [9]. These are typically Internet Service Providers (ISPs) or large enterprises with independent connections to multiple networks. A unique Autonomous System Number is assigned to each AS for use in Border Gateway Protocol (BGP) routing [9]. BGP is an inter-AS routing protocol which interchanges network reachability information with other BGP systems by "advertising a set of of destinations as an IP prefix" [10]. While BGP is also used within an AS (called internal Border Gateway Protocol (iBGP) in this case), we focus on the case of inter-AS connectivity. "Each BGP table is a list of AS paths that packets should traverse from a given router to the prefix containing its destination IP address. The AS terminating an AS path for a given prefix in a core routing table is administratively responsible for this prefix and is called an origin AS" [11]. So we can also think of network topology in terms of examining the ways in which ASes are interconnected. This level of network topology examination is at a far higher level of abstraction than those discussed above but has many important implications. These will be discussed more in depth later.

2.2 Networking and Traceroute Basics

Computers communicate over the Internet via the TCP/IP suite. IP is the principle method of routing packets of data through a network in order to reach their destination [12]. Transmission Control Protocol (TCP) is responsible for the end-to-end connections between the two services that are running on separate computers [13]. Together, they form the TCP/IP suite and allow for connectivity between two nodes on a network and the transfer of data between them [14]. Under the IP, every packet sent over a network contains a time to live (TTL) field, which specifies the maximum number of router hops the packet can traverse enroute to its destination before it should be discarded. The TTL field was originally designed to prevent infinite packet forwarding loops, and routers decrement the TTL of each packet before being forwarded. If the TTL field of a packet reaches zero on its path to a destination, an Internet Control Message Protocol (ICMP) Time Exceeded error is sent back to the host that originated the packet. By explicitly setting the TTL field of packets, each with an incrementing TTL field, an attacker can determine the full path to a node in a network by keeping track of which IP addresses send ICMP Time Exceeded errors.

This process is the basis of the program traceroute, which maps the hops between the source and a specified destination on the Internet. The traceroute program also returns the RTT for each hop, which is the amount of time it takes for a response to a probe to return to the probing machine. It is important to note that this method only reveals the forward Layer 3 path, i.e. the IP addresses that the probes hit on the way from source to destination. If a node has multiple Layer 3 interfaces, traceroute will only reveal the IP address of the interface nearest to the attacker. By iterating through traceroutes to every node on a network, an attacker can attempt to infer the network topology [15]. A typical traceroute command can be seen in Figure 2.2.

```
# traceroute 192.168.1.80
```

Figure 2.2: An example traceroute command.

Suppose the client laptop at IP address 192.168.3.101 in Figure 2.1 ran a traceroute to the web server at 192.168.1.80 with this command. The output would appear as in Figure 2.3.

```
traceroute to 192.168.1.80 (192.168.1.80), 30 hops max, 60 byte packets
 1  192.168.3.100 (192.168.3.100)  2.859 ms  5.490 ms  7.707 ms
 2  192.168.3.1 (192.168.3.1)  9.673 ms  11.539 ms  14.296 ms
 3  192.168.1.80 (192.168.1.80)  16.250 ms  18.176 ms  20.255 ms
```

Figure 2.3: An example of traceroute output.

2.2.1 Types of Traceroute

The default traceroute program on Linux machines employs UDP to send probes. By default, this probe will send packets to a port that is unlikely to be in use so that the destination host does not actually process the packet and simply responds with a ICMP port unreachable message. Specifically, the first probe (with TTL equal to 1) will be sent to port 33434, then the port number is incremented by one for each probe packet thereafter [15]. For Windows machines, the default is ICMP. Additionally, traceroute can be directed to use TCP or to ports commonly used for other services, such as UDP port 53 or TCP port 80. This practice is often employed to circumvent firewalls and is discussed in detail in Section 2.3.

Nmap, the common network scanning tool, has a slightly different traceroute implementation. Nmap's traceroute uses results from its port-scanning probes in order to determine the optimal port and protocol that it should use for its traceroute probe. For example, if nmap detects that TCP port 80 is open on the destination host and allowed through the firewall, it will send its probes to that port. The second interesting methodology that nmap employs is that, instead of starting with low TTLs and incrementing by one with every packet, it begins with a high TTL and works its way down. This allows it to "employ caching algorithms" that "speed up traces over multiple hosts. On average Nmap sends 5-10 fewer packets per host, depending on network conditions. If a single subnet is being scanned (i.e. 192.168.0.0/24) Nmap may only have to send two packets to most hosts" [16].

Traceroute improves its efficiency by sending out probe packets simultaneously. By default, linux traceroute sends out 3 probe packets per hop and 16 probes simultaneously [15]. Having numerous packets in flight simultaneously greatly improves the speed of performing a traceroute but presents the problem of deciphering which ICMP responses correspond to which probes. Essential to the solution of this problem is the concept of ICMP quotations. According to protocol, an ICMP Destination Unreachable message contains the IP header and the first 64 bits of the original IP packet that induced the error message [12]. If the original IP packet contained a protocol that uses port numbers, like UDP, then the port number will appear in those first 64 bits of the returned ICMP Destination Unreachable message [12]. Thus, the traceroute program can parse the returned ICMP message to find the destination port of the original packet and match the response with the probe that it sent. This is the methodology used for the original UDP traceroute program [17]. ICMP traceroutes employs "a unique ICMP id/sequence pair in each outgoing (and thus responding probe)" because there are no ports specified in an ICMP message [17]. If a prober wishes to send their probes to a well-known port, such as UDP port 53 or TCP port 80, it sends each probe with a unique IP-ID value so that it can match the responses [17].

2.3 Traceroute and Firewalls

A firewall is a device or piece of software that regulates what traffic is allowed into or out of a network. Essentially, "a firewall builds a blockade between an internal network that is assumed to be secure and trusted, and another network, usually an external (inter)network, such as the Internet, that is not assumed to be secure and trusted" [18]. Common examples of firewalls include Comodo Internet Security [19] and iptables [20]. Iptables is built into the Linux kernel and allows users to build a customized firewall with user defined rule sets and actions [20]. Iptables' rules define a set of conditions and a set of actions for how to deal with incoming traffic. If an incoming packet matches the conditions of a certain iptables rule, the Linux system will take the action specified by that rule. For example, a Linux web server often has a rule whose condition is to match TCP traffic destined for port 80 and whose action is to accept the packet.

Since the proliferation of firewalls on the Internet, the simple methods of traceroute are often blocked. Packets sent to unknown ports are often summarily discarded when they hit firewalls along with ICMP Echo Requests and error messages. However, there are variations of traceroute that may still work through a firewall. Network administrators are often required to allow traffic headed to certain ports such as UDP port 53 (associated with

Domain Name System (DNS) or TCP port 80 (Hypertext Transfer Protocol (HTTP)) in order to allow for critical services to function. A traceroute user may specify that their probes be sent to these well-known ports because they are usually left open [15]. So, even if a firewall allows only port 80 TCP traffic and blocks everything else, an attacker can still sneak a traceroute through. It is also common for a firewall or router to block all ICMP messages, which results in a traceroute receiving no information about the hops on the path behind that firewall.

Now, for a traceroute deception scheme to be ultimately effective, one must be able to detect and deceive every possible variation of traceroute. Obviously, if the deception is unable to fool a single method of traceroute probing, the whole deception becomes ineffective. Due to the numerous variations of traceroute, many of which were designed for the purpose of avoiding detection, it is difficult to identify whether an incoming packet is a traceroute probe or legitimate traffic. Detection methods will be discussed in detail in Chapter 3.

2.4 Passive Network Mapping

There exist some methods of employing passive measurements to determine topology. The advantage of passive mapping is that the researcher/attacker does not have to send out probes in order to gather information. By its very nature, passive mapping is less likely to cause network disruptions and reduces or eliminates the chance that their mapping activity is detected by network defenses. On the other hand, since the mapper is not choosing the responses that they want to illicit, they have less control over the information that they gather and may obtain less useful information than they would with an active approach.

One example of this type of passive approach would be examining BGP routing tables. BGP tables can be used to map IP addresses to their origin ASes [11]. These tables and updates are easy to process and give an overarching view of how ASes are interconnected and of the topology of the Internet at an AS level. Figure 2.4 depicts a simple set of AS connections. There are three main types of AS connections: customer-to-provider, peer-to-peer, and sibling-to-sibling. Customer-to-provider relationships occur when a smaller AS or ISP agrees to pay another, usually larger, AS or ISP in order to transport traffic that is destined to regions of the Internet that the smaller AS cannot reach. A peering relationship occurs when two ASes, usually of similar size, agree to transport each other's traffic without charge. Peering allows an AS to have access to other parts of the Internet without having to pay a provider AS to transfer its traffic. Sibling-to-sibling relationships are connections between two ASes that are owned by the same company [21]. In Figure 2.4, these relationships are illustrated by the lines drawn between the different ASes, with the direction of the arrow representing the flow of money between ASes. Examining the BGP tables for any of these ASes would reveal this network map. For example, examining AS 2's BGP table would reveal that it is directly connected to AS 1, 3, and 4 and has paths to AS 5, 6, and 7 through AS 3.

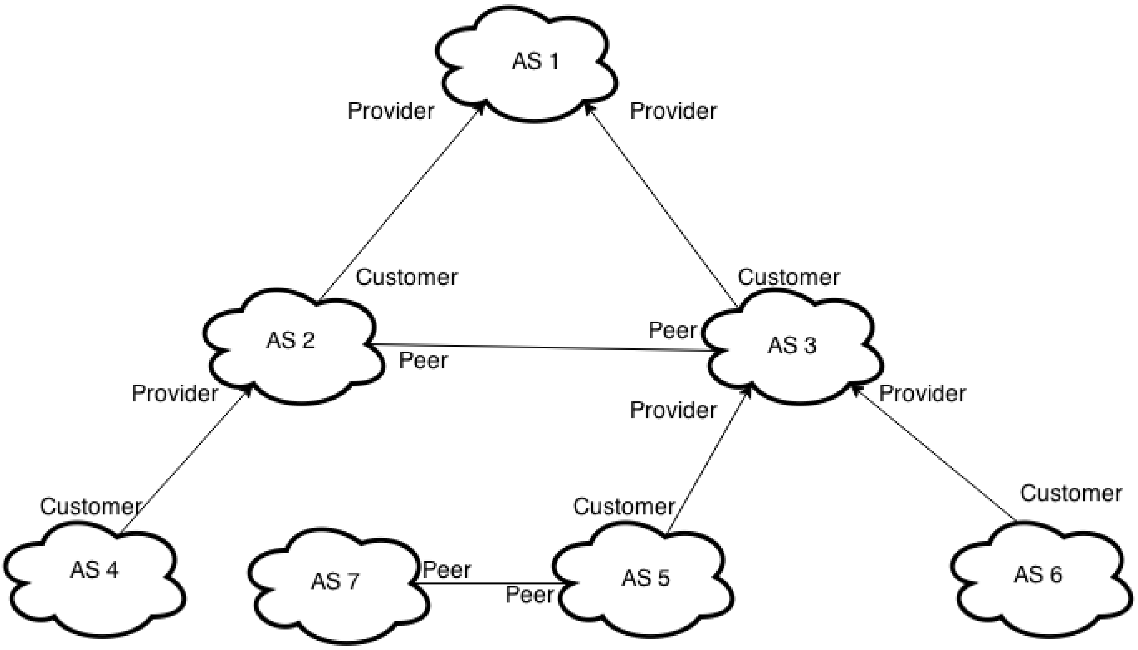


Figure 2.4: An example of an AS topology map. The arrow direction represents the flow of money in exchange for transporting Internet traffic.

Some drawbacks specific to using BGP to examine topology are that "BGP data reflects a control-plane signal rather than how traffic actually travels toward a destination network...tend to capture much less peripheral (not core) connectivity (peering) among regional networks...[and] does not reveal short-term AS path variations and load balancing" [11].

AS connections are essential to the daily operations of the Internet as a whole because they allow for the connectivity between any two endpoints that are not contained within the same AS. There are numerous examples of malicious or accidental misconfigurations which caused whole blocks of the Internet to lose connectivity. For a short period in April 2010, China Telecom advertised 37,000 prefixes, a large increase over its legitimate 40 prefixes. This was likely due to a misconfiguration, but caused connectivity issues for many customers [22]. An instance of a nation state intentionally hijacking prefixes occurred in February 2008, when Pakistan Telecom began advertising part of YouTube's assigned network in an attempt to block Pakistani citizens from accessing YouTube [23]. This false route propagated outside of Pakistan and disrupted service for approximately two hours [23]. These BGP tricks can be used to deny service, reroute traffic through an AS, potentially allowing it to eavesdrop and/or inject traffic that it should never have access to. These are all BGP tricks that are another form of network deception. In each of these cases, someone abused (either intentionally or accidentally) the routing infrastructure in order to deceive both network routers and people as to the true structure of the Internet.

AS connections also have large financial implications as peering relationships dictate the cost of each byte that flows between two ASes. Along that same vein, net neutrality is a concept that ISPs treat all types of data equally, instead of discriminating against content, platforms, or users [24]. One particularly interesting application of our deception work would be to spoof AS interconnections in a way that would represent a degenerate routing situation, where a network probe would travel back and forth between two ASes in a routing loop and see if researchers would notice this phenomena. Another scenario of interest would be to spoof interconnections between the DOD and Star Joint Venture Co., which is the only ISP in North Korea. The limitations that using BGP data for topology mapping, discussed above, make it easier for us to present these deceptive behaviours that show relationships that are not in the BGP tables because people would believe their BGP table view was incomplete. Our deception occurs in the data plane, which is generally considered definitive in terms of how traffic is actually being routed through the network.

Historically, BGP has been used to create deceptive traceroute behavior, most famously when The Pirate Bay claimed they relocated their hosting service to North Korea [25]. When someone ran a traceroute to The Pirate Bay's web server, it responded with a route that ended at an IP address in North Korea. Additionally, The Pirate Bay spoofed an AS path that ended with AS numbers 131279 and 51040. 51040 is The Pirate Bay's AS number and 131279 belongs to STAR-KP Ryugyong-dong, a North Korean ISP [26]. Beyond the fact that The Pirate Bay employed false BGP announcements to advertise a spoofed AS path, we do not know anything about how their deception was implemented as The Pirate Bay has maintained silence on the subject of their deception. Their deception was detected rather easily because they still had to provide normal functionality for users accessing their web page. If a user inspected the RTT of TCP packets sent to The Pirate Bay's web server, they saw the true delay between them and The Pirate Bay's web server. Using these true RTT values, users were able to able to triangulate the web server's true location to somewhere in Germany [27]. In order to implement a truly plausible deception, The Pirate Bay would have had to implement increased delay for connections to their web server. The reason they elected not to do this is probably that it would have affected their user experience negatively and deteriorated the service they provide.

Another weakness in their deception was that they did not provide feasible delay values for certain hops in their deceptive path. While The Pirate Bay implemented some random delay in their deception, there were glaring errors such as two adjacent hops being on opposite sides of the Atlantic Ocean but having very similar RTTs. Additionally, their deceptive path was not plausible in that it bounced back and forth between the United States and Europe several times, traversing the Atlantic Ocean more than two or three times before arriving at its destination. While this deception scheme is similar to what we hope to accomplish, there are a few differences. First, we plan to build a well-documented tool that can be replicated and tested in order to improve current topology measurement systems. Secondly, we do not rely upon BGP shenanigans in order to achieve our deception, which is beneficial because abusing BGP can lead to connectivity issues for large portions of the Internet. Third, we hope to present realistic deceptive paths with realistic RTTs that are based upon source and destination IP addresses so that we may our faked traceroute results appear plausible regardless of the prober's location.

2.5 Uses of Topology Data

Although network topology data is difficult to gather accurately (particularly at the Internet scale), it has many uses and is essential for understanding the "technical, economic, and regulatory aspects of the Internet" [28]. One application of this data is to study the growth of the Internet over time and how the implementation of novel technologies (such as IPv6 or Classless Interdomain Routing) affects the way that the world is interconnected and how information travels across the globe.

Another application of topology data is to study AS relationships in order to better understand the economic underpinnings of the Internet. Understanding how ASes connect to each other and the type of relationship they have (settlement-free, parent-child, peering) can lend itself to understanding how our packets actually get routed through the Internet and what that process entails. The fact that Internet traffic is constrained by these AS peering relationships implies there are only certain paths over which information can flow through our networks, which influences strongly how networks function and evolve over time. A view of the network topology and performance is incomplete if we do not consider the economics that influence how the Internet works.

2.6 Network Mapping Defense

A solution that many well-defended networks employ to prevent this mapping attack is to simply drop any network probing packets that are detected [29]. While this is surely a superior solution to allowing an attacker an accurate mapping of the network's topology, it does leak information to the attacker, namely that there is a network behind that router that warrants protection. Additionally, sending no reply whatsoever robs the defender of the opportunity to intentionally mislead the attacker, keep them in an intelligence-gathering phase, and potentially thwart future attacks with deception. An alternative approach has been proposed: present the attacker with a false network topology in reply to network probes. While there has been some work in this area, the idea is still in its infancy and does not exist beyond a simple proof of concept implementation [3]. If this methodology does prove viable it will be a valuable addition to any network administrator's defense efforts.

2.7 DNS Deception

Another source of topology deception is the abuse of DNS records. A DNS A record maps a hostname to a 32 bit IPv4 address, which is what we commonly think of when we think of DNS. However, DNS can has many additional capabilities and can store a wealth of information. Another example of a DNS record is a PTR record. This record is used for reverse DNS lookups, i.e., when a user has an IP address and wants to know the canonical name associated with that address. The problem with DNS records is that they are not authenticated or verified in any way. Anyone with access to a DNS server can rewrite the DNS names for any IP address for which they are the authoritative DNS server. An attacker could not spoof DNS records for any IP address on the Internet, only the records whose IP addresses their server is responsible for. Essentially, given a set of IP addresses, an attacker could make their domain names anything they desired. However, given a set of domain names, an attacker could not choose random IP addresses for those domain names.

A fun, non-malicious example of this would be the system administrator who used an unused /24 network and Reverse DNS pointer records to write the introduction web crawl from Star Wars Episode IV to the screen when someone runs a traceroute to his network. The output of a traceroute to 216.81.59.173 is shown in Figure 2.5. This is a real-world example of deception that is currently live on the Internet at the time of this writing. It was deployed using virtual routing and forwarding (VRF) on two Cisco routers and spoofed DNS records for an unused /24 subnetwork. VRF is a piece of software that allows for multiple routing tables to coexist on the same router. Let us label the two Cisco routers router A and router B. Any time an incoming packet destined for the IP address 216.81.59.173 arrives at the border router (router A), it is forwarded to router B. Router B then has a VRF that forwards the packet back to router A. The incoming probes are forwarded back and forth between the two routers in order to lengthen the deceptive path. This is how the additional hops are introduced into the path. Since the person that implemented this deception had access to the authoritative DNS server for this /24 network, they were able to update the DNS PTR records to match the print the Star Wars opening crawl from the Star Wars Episode IV movie [30].

```
traceroute to 216.81.59.173 (216.81.59.173), 30 hops max, 60 byte packets
...
 12   145 ms  Episode.IV [206.214.251.1]
 13   160 ms  A.NEW.HOPE [206.214.251.6]
 14   174 ms  It.is.a.period.of.civil.war [206.214.251.9]
 15   172 ms  Rebel.spaceships [206.214.251.14]
 16   168 ms  striking.from.a.hidden.base [206.214.251.17]
 17   166 ms  have.won.their.first.victory [206.214.251.22]
 18   175 ms  against.the.evil.Galactic.Empire [206.214.251.25]
 19   158 ms  During.the.battle [206.214.251.30]
 20   167 ms  Rebel.spies.managed [206.214.251.33]
 21   164 ms  to.steal.secret.plans [206.214.251.38]
 22   161 ms  to.the.Empires.ultimate.weapon [206.214.251.41]
 23   161 ms  the.DEATH.STAR [206.214.251.46]
 24   159 ms  an.armored.space.station [206.214.251.49]
 25   157 ms  with.enough.power.to [206.214.251.54]
 26   137 ms  destroy.an.entire.planet [206.214.251.57]
 27   165 ms  Pursued.by.the.Empires [206.214.251.62]
 28   172 ms  sinister.agents [206.214.251.65]
 29   170 ms  Princess.Leia.races.home [206.214.251.70]
 30   168 ms  aboard.her.starship [206.214.251.73]
 31   161 ms  custodian.of.the.stolen.plans [206.214.251.78]
 32   176 ms  that.can.save.her [206.214.251.81]
 33   161 ms  people.and.restore [206.214.251.86]
 34   157 ms  freedom.to.the.galaxy [206.214.251.89]
...
 59   165 ms read.more.at.beaglenetworks.net [216.81.59.173]
```

Figure 2.5: An example of tricks with DNS resolution.

Our deception is different in that we create spoofed ICMP messages instead of using VRF to induce routers to create the deception for us. Additionally, we do not deploy any DNS spoofing, which was unnecessary for us because we are able to spoof paths that include IP addresses for which we do not have access to the authoritative DNS server. Their deception methodology only contained IP addresses within their assigned IP space. Our deception methodology allows us to implement paths that contain any IP addresses, which increases our flexibility in deploying deception. Another weakness in the Star Wars deception is the constant RTT values of the traceroute probes. Because they do not introduce any artificial delay into their responses, the RTT remains relatively constant for each hop in their deceptive path. This makes the deception easy to detect. However, their end goal was to create an entertaining trick and not to actually deceive a prober [30]. Our end goal is to create a plausible deception that will fool traceroute probers.

While this is a funny trick, it does show that there is no authentication of DNS records in any way. So, we can create any name to IP address mapping, within our authoritative IP space that we desire in order to create a more complete deceptive topology.

We can do certain things such as using domain names of known companies like Comcast or AT&T and simulate peering relationships that do not exist. We could even introduce delays in these apparent peering relationships that could damage the reputation of one or more telecommunications companies.

It is also common for large ISPs to name routers and hosts in a way that conveys information about the network topology or geographic locations. Sometimes, it is possible to tell the cities that a traceroute probe travels through simply by doing reverse DNS lookups on all the hosts that the probe hits on the way to its destination. It is possible for us to exploit these naming conventions to make it look like the network path to a destination bounces around several cities, when really each of these hops is contained within one wiring closet [31].

The lack of authentication and integrity of DNS records and messages means that we can abuse it in order to create a more thorough and robust deceptive behaviour.

2.8 Alias Resolution

One key shortcoming of using traceroute to perform topology mapping is that traceroute gives an interface-level view of the network. However, each router has at least two interfaces and are likely to have many more. While each interface will have its own IP address, traceroute will only reveal a single IP per router. If we harbor any hope of drawing an accurate topology map, we have to deconflict these IP addresses. The process of discovering which IP addresses are actually separate interfaces on the same router or device is called IP alias resolution [32]. There has been an abundance of work in this area, but it remains a difficult problem with no sure solution in sight. Alias resolution difficulties actually contribute to our deception abilities because mapping a network is already such a hard problem that any deceptive results are likely to be believed out of hand.

CHAPTER 3: METHODOLOGY

Our end goal is to create a tool that can be deployed at a network's ingress routers that responds to traceroute probes in a deceptive manner without affecting normal, legitimate traffic destined to the network. We believe that this network topology deception would be complementary to other defense techniques such as LaBrea Tarpit and IP Personality that attempt to deceive attackers in order to bolster the security of the defended network [33], [34]. LaBrea operates by taking over unused IP addresses and creating virtual hosts that an attacker would consider an attractive target. Additionally, it uses TCP flow control to limit the window size of incoming TCP connections in order to slow down attackers and worms. This means that the attacker will waste time and effort attacking a host that does not exist on the network, which reduces the efficacy and profitability of his attack [33]. IP Personality modifies the packets being sent by an existing system in order to make it appear as though the system is running a different operating system than it is. Since each operating system has unique exploits to which it is vulnerable, IP Personality can aid in security by causing an attacker to attempt attacks that are ineffective against the operating system the system is actually using [34].

We imagine our network topology deception scheme to have similar benefits as these techniques. Additionally, we conjecture that DeTracer could be used in tandem with these other deceptive tools in order to create host and network honeypots and sinkholes that are more realistic. When deployed together, we could create honeypots that have dedicated subnets as well as their own routes leading to them with the end goal of creating a more realistic deception in order to improve the security of our legitimate hosts and network links. While none of these tools is a replacement for secure, patched systems and firewalls, they increase the amount of work that an attacker must undertake in order to exploit a network and potentially thwart attacks by directing the attacker down fruitless avenues.

Our goal with this research is to create a deceptive topology tool that has the following properties:

- Robust traceroute probe detection
- Ability to create complex, interconnected topologies
- Simulate the appearance of load-balancing routers and multipathing
- Variable path lengths to real and faked end hosts
- Unique deceptive behaviors based on the source and destination IP prefix
- Create realistic RTT for traceroute probes
- Easily configurable false network topology

In order to advance the concept of presenting a false network topology to network probes, we implement behavior on the basis of source and destination prefixes of incoming packets with the end goal of presenting an interconnected faked network. Without implementing per-destination network specific behavior, we cannot reveal multiple paths or additional deceptive complexity within the network. If we do not base the path presented on the destination of the probe, we are limited to presenting a single path as seen in Figure 3.1. Note that from this point on, when we refer to a router as router X, we mean the interface on that router that would respond to a traceroute probe. Clearly, this topology is far too simplistic to accomplish our deceptive goals.

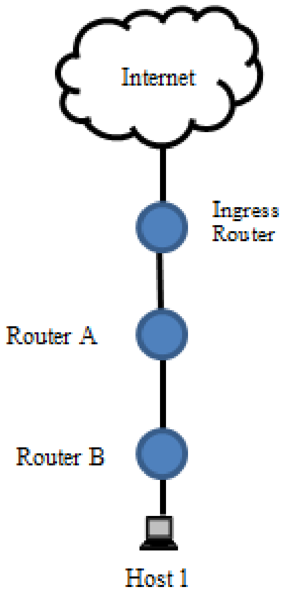


Figure 3.1: An example of a faked topology with a single path.

Additionally, we cannot simply generate a random path for each end host in our fake topology. This behavior would result in a picture like the one in Figure 3.2, where each traceroute probe is presented with a linear path to the end host that is completely disjoint from the paths to other hosts on the network. Also, we would like the ability to present paths of varying lengths to different subnets of our network.

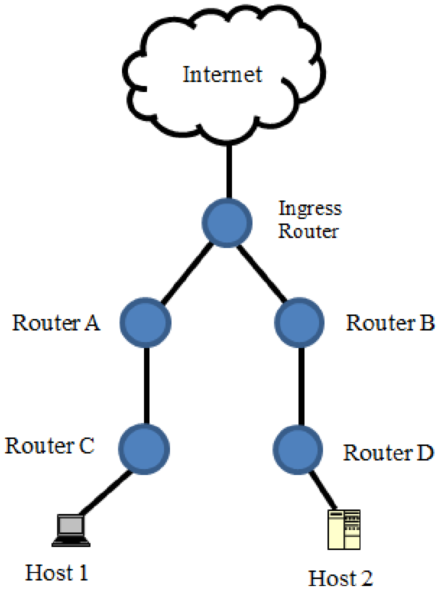


Figure 3.2: An example of faked topology with only linear paths.

While this is a feasible network topology, we would like to be able to present more complicated false topologies. Real-world networks are far more interconnected with redundant paths and numerous links between routers on the path to an end host. However, it is also possible that the real topology of our network is linear in nature, which makes it an attractive target to an attacker. A network without redundancy is far more vulnerable because the failure of a single link results a complete loss in connectivity between end hosts. The capability to disguise this weakness by presenting an interconnected network topology would be a key benefit of our deceptive model. If we modify the deceptive behavior on the basis of the destination address of the mapping probe, we will be able to present a more interconnected network by grouping the hosts into subnetworks and allowing virtual routers that are in a path to a certain subnet to appear in paths to other subnets. This will simulate the existence of load-balancing and redundancy even in environments where the real topology does not have such safeguards.

An example of a more interconnected network is seen in Figure 3.3. In this example, the hosts are grouped into subnets based upon their IP address prefixes. The first 24 bits of each host's 32-bit IP address are all identical to identify that they belong to the same subnet and the last 8 bits of their IP address identify individual hosts on the subnet. All the hosts on the 13.37.100.0/24 subnet share the IP prefix 13.37.100.0 and all the hosts on the 13.37.200.0/24 subnet share the prefix 13.37.200.0. The remaining 8 bits differentiate hosts on that subnet from each other. This topology would appear more realistic and have a greater chance of successfully deceiving an attacker. The network also appears more resilient to certain attacks like Denial of Service because of the built in redundancy it has in case of link failure. If Router A in Figure 3.3 goes down, there is still a path from the end hosts out to the Internet. If Router A in Figure 3.2 is attacked, host 1 will no longer be able to access the Internet. In summary, the complexity and design of our fake network topology is essential to creating a deception that will successfully elude an attacker.

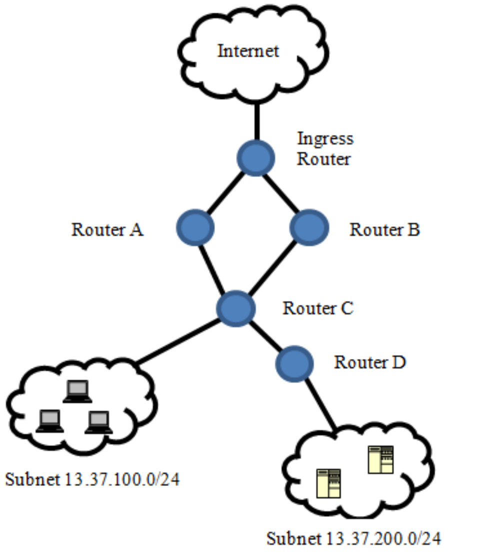


Figure 3.3: An example of interconnected faked topology.

Another important consideration when creating this deceptive behaviour is the RTT and packet loss that the attacker sees when sending traceroute probes. Realistic packet delay and loss are paramount to implementing a believable deceptive network. If there are characteristics like a monotonically increasing delay for each hop with no randomness or variance to the RTT values, a sophisticated attacker will detect that something is out of the ordinary and our methodology becomes easier for a sophisticated attacker to detect.

Last, we need to develop a robust method of detecting network probes in order to enable us to defend our networks against any method of mapping. Since traceroutes appear in many forms, (TCP to port 80, UDP to port 53 or ICMP) we must be able to detect all of them in order to deploy an effective deception. The one commonality that all the variants of traceroute rely upon is the ICMP Time Exceeded messages returned by hops along the path to the destination.

3.1 Methods of Traceroute Detection

In order to induce ICMP Time Exceeded messages, an attacker must send packets with TTLs that will expire along the path to the destination. We have two viable options for detecting traceroute probes at our border router. Our first option is to inspect outgoing traffic for Time Exceeded Messages that originate from inside our network. This method is advantageous because it is efficient and easy to implement. The only work the deception program would have to do in this scenario is intercept ICMP Time Exceeded Messages and overwrite the source IP address of the packet. This rewriting is trivial and would allow for the creation of fake topologies.

There are three main drawbacks of this detection method. First, if there is no host at a certain IP address on our network, packets sent to that IP address will not generate any return traffic. Because our router would be basing its deception on traffic leaving the defended network, we could not create fake end hosts on our network without actually deploying a real machine at that IP address. Additionally, our deceptive topology cannot present a path to a host that is longer than the actual path to the host if we rely on rewriting probe responses. Once a traceroute probe increments to a TTL that actually reaches the host, that host will respond with a port unreachable or other message that will confirm its existence. Since we have no certain method of detecting what outgoing traffic is actually a response to a traceroute probe, we cannot intercept this traffic. Consequently, once the probes begin to actually reach the destination host, we no longer can create spoofed ICMP Time Exceeded messages. Finally, this method allows for more traffic to be created on our internal network instead of having it all be handled by the ingress router. While the amount of traffic generated by traceroute probes is generally minimal and unlikely to cause any network disturbance, it is another element to consider.

An alternative, superior method of detecting traceroute probes is to inspect incoming traffic at our border router. We can use a TTL filter in order to detect packets that would potentially expire inside our network. We accomplish this by implementing an iptables rule on our ingress router that will forward all traffic with an TTL less than a certain value to the our deceptive program. This value is based upon the network depth of the fake topology, where network depth is defined as the maximum number of hops that a packet can take within our network from the ingress router on which the deception is running.

The advantages of this approach are that we can respond to traceroutes destined to IP addresses that do not actually exist on our network. We can also create a path of any length, so long as the number of hops in our fake path is less than the maximum TTL of packets that we are intercepting. For example, if we are intercepting all packets with TTLs less than 30, we can create a fake path to a destination that is 30 hops from the border router. On the downside, we are forced to intercept far more traffic than we were in the first method. Additionally, we have a greater chance of intercepting legitimate traffic with this filter because we are filtering everything with a low TTL, not just Time Exceeded messages. However, almost all operating systems default to a TTL of 64 or greater [35]. Huffaker *et al.* found that most end to end paths on the Internet are relatively short with virtually all the measured IP path lengths containing less than 30 hops [11]. Thus, we expect it to be unlikely that any legitimate traffic to be intercepted by our deception program. However, we would like to see this claim investigated further in future work. Finally, this methodology results in our deception program having to do far more processing. The deception program must not only handle more packets than it does in the first method, it has to interpret the incoming packets, determine whether they would expire within the fake network, and, if so, craft the response packet from scratch while also incorporating a realistic amount of delay that is proportional to the TTL of the incoming packet.

3.2 Deception Program Details

The deception program is written in Python and primarily makes use of the Scapy library in order to read and write packets. Scapy is a “packet manipulation program” that can capture and forge packets [36]. We use the nfqueue-bindings library to act as an intermediary between the Linux kernel packet handler and the Scapy program running in user space. nfqueue-bindings essentially transports packets from kernel space to user space and back, allowing us to interact with the packets using a high level language like Python [37]. Nfqueue-bindings creates an iptables chain, called NFQUEUE, which will place packets into a queue to be picked up by the Scapy program. Scapy must issue a drop or accept verdict for each packet to clear it from the queue [38]. We elect to use Scapy to parse and craft packets because it is easy to use and configure to our needs. It also allows us to use the vast functionality native to the Python language that allows us to present more complex topologies and implement behavior based upon the source and destination IP addresses of probes. An example of a false network topology that DeTracer can present is depicted in Figure 3.4.

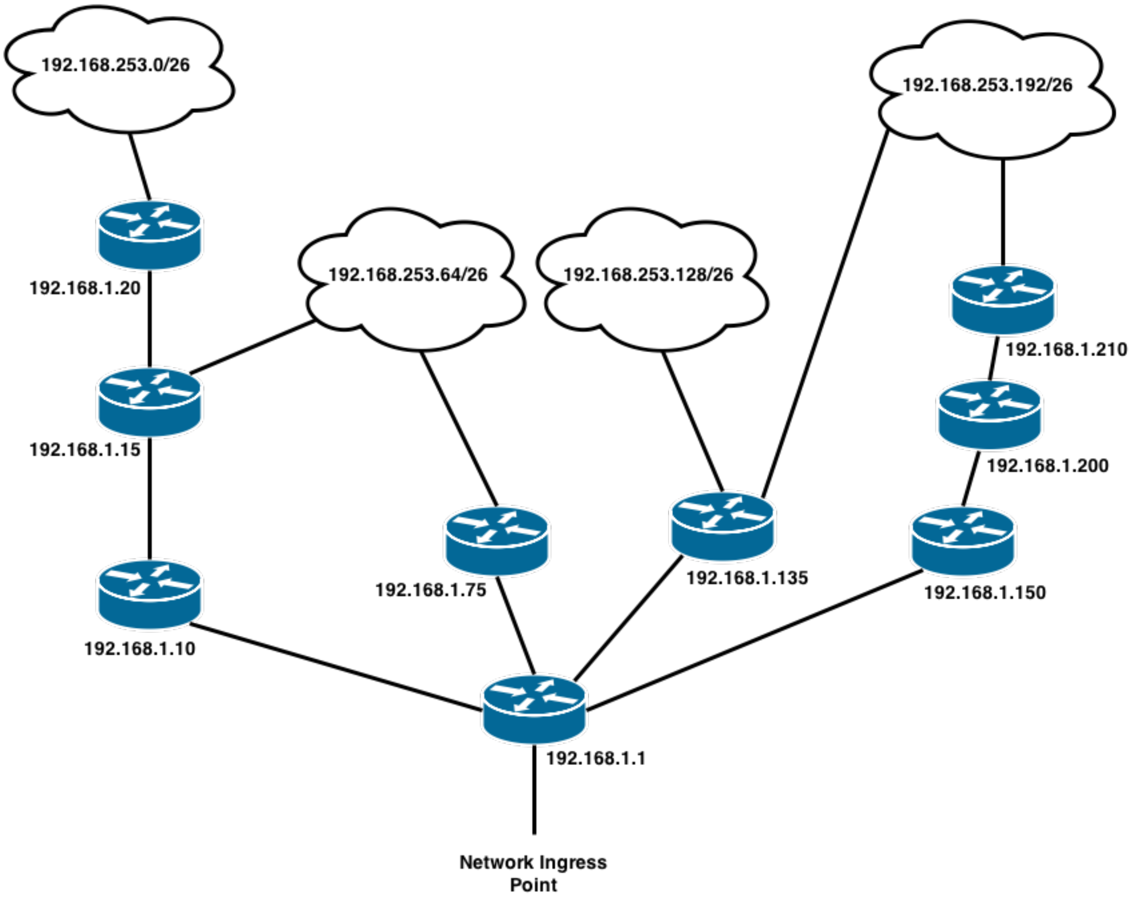


Figure 3.4: An example of a faked topology deployed by DeTracer. Note that the IP addresses correspond to responding interface on each router.

DeTracer's logical flow can be seen in Figure 3.5 and is described as follows. Initially, the deception program runs a configuration script that ingests the fake topology as a Python graph object, sets up the proper iptables rules to forward expiring TTL traffic to the NFQUEUE chain in order to be queued and dealt with by the Scapy program. In this graph object, IP addresses are specified as strings ("192.168.1.1") and subnets are specified as strings in Classless Inter-Domain Routing (CIDR) format ("192.168.1.0/24"). The configuration script also generates two radix tree objects and passes them to the deception script. One radix tree object contains the intermediate router in our fake topology and the path a packet would take to each router. The routers are represented as IP addresses in string format and the paths are represented as a list of IP address strings. The second radix tree contains all the subnets in our fake network and the path through the intermediate routers a packet must take to reach that subnet. The subnet is stored as a string in CIDR format ("192.168.1.0/24") and each of the paths is represented by a list of IP addresses in string format.

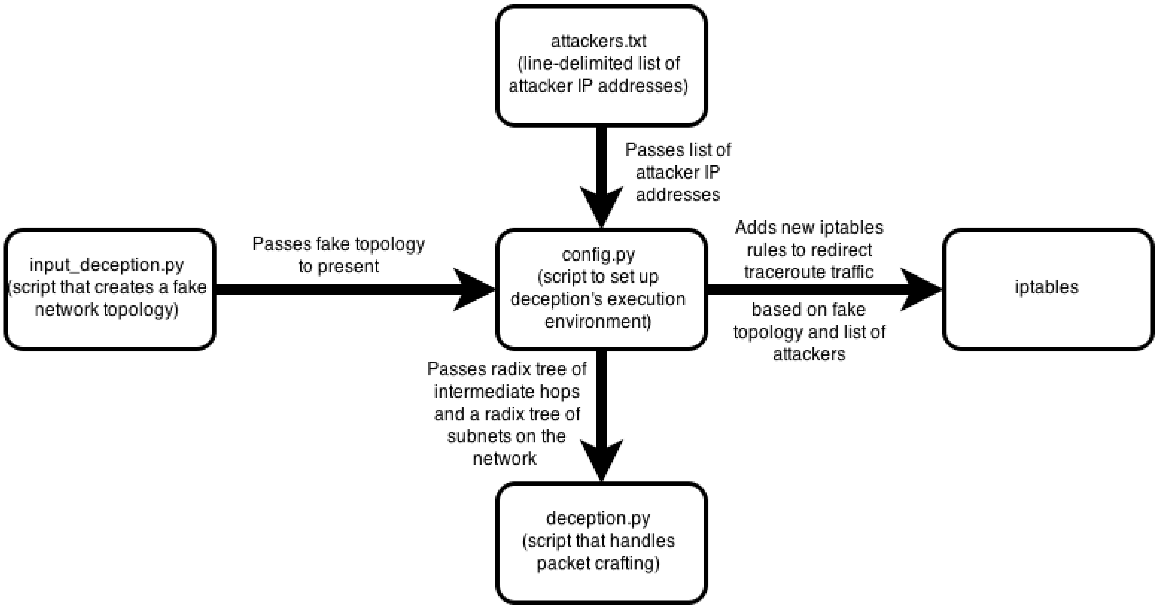

attackers.txt
(line-delimited list of attacker IP addresses)
Passes list of attacker IP addresses
input_deception.py
(script that creates a fake network topology)
Passes fake topology to present
config.py
(script to set up deception's execution environment)
Adds new iptables rules to redirect traceroute traffic
based on fake topology and list of attackers
iptables
Passes radix tree of intermediate hops and a radix tree of subnets on the network
deception.py
(script that handles packet crafting)

Figure 3.5: DeTracer's logical flow control.

Figure 3.6 shows an example iptables rule for redirecting incoming packets with a TTL less than 30 to the NFQUEUE chain. DeTracer's configuration script runs the equivalent of this command.

```
# iptables -I FORWARD -m ttl --ttl-lt 30 -j NFQUEUE
```

Figure 3.6: A sample iptables rule for redirecting traceroute traffic to the NFQUEUE chain.

The minimum TTL allowed into the network by the ingress router is determined by the fake topology, with the value being selected as the longest path in the fake topology graph. A second approach to determining the cutoff value for TTL values of incoming packets would be to investigate the traffic at our ingress router. If we can determine the minimum TTL for non-traceroute traffic at our particular router, we can intercept all incoming packets with a TTL less than that value and be sure we are not interfering with normal traffic on the network.

In our deployments, we chose to intercept packets with TTL values less than or equal to 30 at our ingress router where DeTracer was deployed. We chose this value because it was unlikely to interfere with non-traceroute network traffic while still intercepting the vast majority of traceroute probes. By default, traceroute will stop probing after 30 hops, meaning that the packet with the highest TTL sent by a default traceroute probe will have a TTL of 30 [15].

Once a packet is picked up off the queue to be handled by Scapy, it is handed to the sendSpoofedResponse() function which is seen in pseudo code in Figure 3.7. Essentially, the destination IP address and TTL values are examined to determine where in the fake topology that packet would expire, if at all. First, we check if the packet is destined for one of the routers on our fake network by searching the radix tree of routers. If it is, we obtain the TTL value of the incoming packet using Scapy's built in packet parsing functionality and compare it to the length of the path to that router.

```
def sendSpoofedResponse():
    # get the data from the incoming packet
    pkt = incoming_packet
    drop(incoming_packet)

    # Check if incoming packet was destined for an intermediate
    # router or a fake host with an active IP address
    if (md5(pkt.dstIP)[0] != 0 and \
    routerTree.search_best(pkt.dstIP) == None):
        return

    # introduce some delay based on the incoming packet’s TTL
    time.sleep((pkt.ttl + random.random())/64)

    # determine which path we should reply with
    destinationSubnet = subnetTree.search_best(pkt.dstIP)
    path = destinationSubnet.data[’path’]

    # Assign the source of the response packet according
    # to which hop along the path it would expire
    if (0 < pkt.ttl <= len(path)):
        fakeSrcIP = path[pkt.ttl-1]
        fakeICMPType = 11 # ICMP Time Exceeded
        fakeICMPCode = 0 # ICMP TTL Expired in Transit
    # For the destination packet if incoming probe is ICMP,
    # Craft an Echo Reply
    else if pkt.proto == ICMP Echo Request:
        fakseSrcIP = pkt.dstIP
        fakeICMPType = 0  # ICMP Echo Reply
        fakeICMPCode = 0 # ICMP Echo Reply
    # For the destination packet, assign a port unreachable message
    else:
        fakeSrcIP = pkt.dstIP
        fakeICMPType = 3 # ICMP Destination Unreachable
        fakeICMPCode = 3 # ICMP Port Unreachable

    # build the response packet
    response = IP(dstIP = pkt.src, src= fakeSrcIP)/ \
            ICMP(type = fakeICMPType, code = fakeICMPCode) / \
            IPerror(str(pkt))
    # send the response
    send(resp)
```

Figure 3.7: Pseudo Code for the deception program’s packet crafting function.

If the TTL is less than the length of the path to the router, we create an ICMP Time Exceeded message with the source IP address equal to the hop along the path where the incoming probe would have expired. If the TTL is greater than or equal to the length of the path to the router, we create an ICMP Port Unreachable message with the source address equal to the router that the probe was headed to. If the packet is not destined for one of our routers, we search the radix tree of subnets in order to find which subnet the packet is headed to. We then compare the TTL of the incoming packet to the length of the path to the subnet that it is headed toward. If the TTL is less than the length of this path, then we craft an ICMP Time Exceeded message from the hop at which the packet would expire in our fake topology. If the TTL is greater than or equal to the length of the path to this subnet, the deception will either return an ICMP port unreachable message (if we want that IP address to appear active on the network) or send no response whatsoever (if we want no host to appear at that IP address).

The deception program determines whether a host should be up by taking the MD5 hash of the packet's destination IP address. If the first bit or bits of this hash matches a certain set of values, that destination IP will act as an active host and send a response. We use a hash function because it allows us to have a fast, easily tunable parameter for determining how many hosts on our network will respond to traceroute probes. A bit may take the value of 0 or 1. If we choose to send a response from all hosts whose IP addresses hashes to a value with a first bit equal to 0, then exactly half of the IP addresses on our fake network will appear to be active and responding to traceroute probes. If we only send responses from IP addresses whose hash value begins with the bit sequence 00, then exactly one fourth of the IP addresses on our network will appear to be up. Additionally, since the same inputs hash to the same value every time, the IP addresses that are up are constant over time. This creates a consistent set of active hosts on the network without having to save the state of every single possible host on the network. This behavior simulates what happens when an attacker runs a traceroute to a host that does not exist, it will receive no response and eventually time out until the attacker stops sending probes to that address.

CHAPTER 4:
FINDINGS

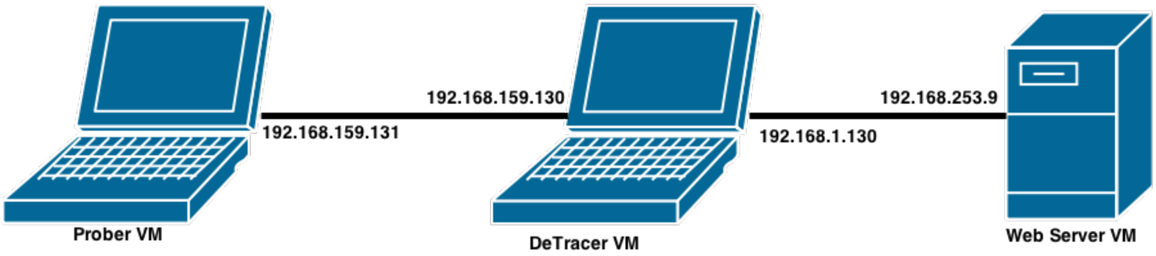
This chapter will discuss the results of our experiments with DeTracer and our selected topologies.

4.1 Experiment Overview

To test and characterize DeTracer, we conducted a series of three experiments. Initially, we deployed DeTracer in a network of virtual machines (VMs) running on a single physical host as a proof of concept and convenient testing platform. Next, we ran DeTracer directly on a physical machine and probed it using separate physical machines directly connected via the local subnetwork. Finally, we deployed DeTracer on a publicly reachable machine on the Internet. In this instance, DeTracer was running on a virtual machine with a globally routable IP address.

4.1.1 Virtual Machine Setup

The first setup we used was a single Windows 7 host running VMWare Workstation. The goal of this experiment was to verify that the router running DeTracer was able to present a deceptive network topology without impeding access to the web server's web page. Within Workstation, we ran three separate VMs: i) a prober; ii) a router running DeTracer; and iii) a web server. The configuration is depicted in Figure 4.1.



.

Figure 4.1: The logical layout of the three VMs in the virtual network setup.

The router VM had two interfaces, each of which was on a separate network. The prober VM was on one network and connected to one interface on the router VM. The web server

VM was on the second network and connected to the other interface on the router. Both the web server and the prober VM had the router VM as their default gateway. The router VM ran Fedora 19 (Linux kernel version 3.9.5 and iptables version 1.4.18) with DeTracer running on it. The web server was a Ubuntu distribution running Apache while the prober VM was another Fedora machine. We found that installing the perquisites for running DeTracer was easiest on a Fedora machine. The choices for the other OSs was arbitrary. Although we found DeTracer easiest to deploy on a Fedora machine, we expect DeTracer to successfully work on any Linux-based installation, and that our results were not impacted by the choice of Linux distribution.

In the experiment, the prober VM attempted to perform a network mapping attack on the network behind the router VM on which the web server resides using traceroute and nmap. The prober VM sent UDP, ICMP, and TCP traceroute probes and nmap probes to randomly selected IP addresses that lay in the network behind the deceptive router.

In our experiments, the router VM was able to intercept and spoof responses to the traceroute traffic while forwarding on legitimate HTTP traffic. The prober VM was presented with a false topology when it attempted to map the network but was also able to access the web page served by the web server on the target network. When an attacker performed a traceroute to the web server, DeTracer returned the deceptive path to the web server's IP address. These results gave us hope that it was feasible to deploy DeTracer on a network and maintain normal network function.

Additionally, we were able to achieve several key aspects of our deception methodology. First, we were able to present different routes based upon the source and destination IP address of the traceroute probes. We could also implement blacklist schemes where only known attacker IP addresses were presented with the deceptive topology. Conversely, we could deploy a whitelist scheme where known good source IP addresses were presented with the true network topology instead of the false topology. In other words, we could deploy DeTracer with an implicit deny policy (present the deceptive network topology by default) or an implicit allow policy (present the true network topology by default). Another key finding was that we were able to deceive every method of traceroute that we tested. UDP, TCP, ICMP, paris-traceroute, scamper, and nmap traceroute probes all reported the false network topology that DeTracer presented. Regardless of the port or protocol that the traceroute probes employed, the results returned were the deceptive routes. This proved the efficacy of our traceroute detection methodology and that the TTLs of the incoming packets were enough to detect all traceroute probes. The fake topology presented by DeTracer in this experiment can be seen in Figure 3.4.

This experimentation did reveal some weaknesses with our implementation of DeTracer. We hoped to present realistic RTT values to a traceroute, where the RTT value increased as the TTL of the traceroute probe increased. Additionally, for the RTT values to be realistic, there must be some randomness added into the RTT values an attacker sees, instead of monotonically increasing delay for each hop in the traceroute path. The RTT delays that we saw in this experiment were not feasible for several reasons. We noticed that the RTTs that traceroute was reporting were monotonically increasing for each probe sent. An example output can be seen in Figure 4.2.

```
traceroute to 192.168.253.9 (192.168.253.9), 30 hops max, 60 byte packets
 1  192.168.159.130  0.955 ms  0.330 ms  0.577 ms
 2  192.168.1.1  61.735 ms  124.132 ms  195.002 ms
 3  192.168.1.10  269.623 ms  339.986 ms  408.200 ms
 4  192.168.1.15  494.400 ms  585.285 ms  675.153 ms
 5  192.168.1.20  791.443 ms  907.907 ms  1020.170 ms
 6  192.168.253.9  1161.114 ms  1304.953 ms  1447.516 ms
```

Figure 4.2: An UDP traceroute to a fake host in VM scenario where DeTracer introduces unrealistic monotonically increasing delay. Each probe's RTT increases by approximately 70ms, even when the probes have the same TTL value.

In this example, at attacker at IP address 192.168.159.131 performed a UDP traceroute to the destination IP address 192.168.253.9. The first hop in the trace corresponds to the real interface on the router VM that the prober VM was connected to. Each subsequent hop was created by DeTracer and was fake. The key shortcoming that appeared in this trace was that each probe's RTT incremented by approximately 70ms. A traceroute sends 3 probes per TTL and it is expected that each of these 3 probes have similar RTT values because they traverse the same distance across a network. We saw inconsistencies between probes for a given hop. For example, if we examined the RTT for hop number 3, we saw RTTs of 269ms, 339ms, and 408ms. In a true topology, we would expect these RTTs to be around the same value for each probe with the same TTL instead of these monotonically increasing results. While it would not be unusual in a real network for a given hop to have different

RTT values, it was clearly wrong for every single hop in a traceroute to exhibit the same monotonically increasing behavior. We should have seen delays that were increasing as the TTL increased, with some additional random variance between probes with the same TTL value. It is important to note here that at this point of our experimentation, we did not introduce any artificial delay intentionally. The increased delays for each probe was a result of processing delay native to the DeTracer program. The fact that these RTTs were increasing for each probe and not just as the TTLs increased, along with the fact that the delay increase is almost constant, made the deception scheme easy to detect because the traceroute results were infeasible. The unrealistic delay results were seen regardless of the type of traceroute used, with the exception of nmap's traceroute, which saw infeasible RTTs values in a different way.

An nmap traceroute run against DeTracer revealed that the first hops with lower TTLs had longer RTTs than the hops with higher TTLs. This is caused by the fact that nmap sends probes with high TTLs first and decrements the TTL for each probe. Figure 4.3 depicts an example of nmap's traceroute results directed at DeTracer. Clearly, the RTTs of the first hops should not have been consistently larger than the RTTs of hops that appeared later in the trace. In a legitimate network, nmap traceroute results returned higher RTT values for hops later in the path. These anomalous RTT results were caused by the way that the incoming probe packets were queued and handled by DeTracer in the order that they arrived at the DeTracer host.

```
TRACEROUTE (using proto 1/icmp)
HOP RTT       ADDRESS
1   0.93 ms   192.168.159.130
2   925.11 ms 192.168.1.1
3   823.92 ms  192.168.1.10
4   142.65 ms   192.168.1.15
5   67.34 ms 192.168.1.20
6   1.27 ms 192.168.253.9
```

Figure 4.3: An nmap traceroute to a fake host in VM scenario where DeTracer introduces unrealistic increasing delay. The RTT values decrease as the hop values increase, which is the opposite of what an nmap traceroute should return.

Our theory was that these delay problems were caused by limits of the CPU of the VM running DeTracer and the delay caused by bringing the packets up to user space. Additionally, Scapy had to issue a decision on each incoming traceroute packet, which introduced an increasing queuing delay for each additional packet that arrived at the DeTracer host. We noticed that the Python process associated with DeTracer was running with close to 100% CPU usage.

In order to resolve these issues, we attempted to implement load-balancing by running several separate DeTracer processes, each attached to a separate nfqueue queue of incoming traceroute packets. However, this did not improve the performance of DeTracer and we saw the same monotonically increasing RTTs results as without the multiple queues. Nfqueue implements load-balancing on a per-flow basis, instead of a per-packet basis. This meant that when we had several queues for nfqueue to place packets, it would end up placing a batch of packets in the first queue while the others remained idle. Then, when the next batch of packets arrived, they would be placed in the second queue while the others remained idle. Since we were unable to deploy a more granular, per-packet, load-balancing, we saw no performance gains from implementing load balancing in DeTracer when probing from a single source IP address. In order to determine if the delay issues were caused by limitations in computing resources, we installed DeTracer with a single queue on a physical machine instead of a VM.

4.1.2 Physical Machine Setup

The next phase of our experimentation was to setup a physical machine running DeTracer with more computing resources. The layout of this experiment is depicted in Figure 4.4.

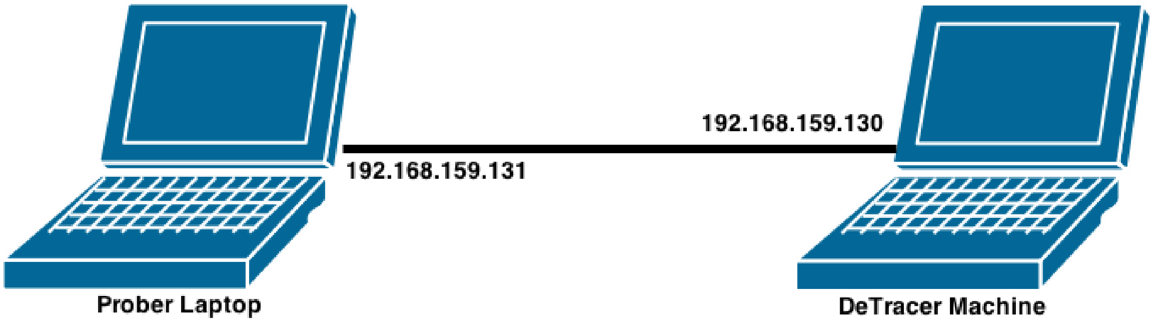


Figure 4.4: The logical layout of the DeTracer host and the prober host in the physical machine setup.

We installed DeTracer on a Dell desktop machine with 4 3.10GHz processors running Fedora 20. This machine had one network interface, which we connected to a laptop using a crossover Ethernet cable. Technically, the desktop running DeTracer was not actually functioning as a router because it did not have at least two interfaces connected to two separate networks. However, DeTracer was still able to present the appearance of a full network topology behind the the desktop as if it were a router. A laptop was used to send traceroute probes to the desktop and inspect the responses.

This phase of experimentation also produced heartening results. Increasing the computing resources available for the DeTracer program solved the problem of monotonically increasing RTT values for traceroute in its default operational mode(s). DeTracer was still able to deceive every method of traceroute that we tested. Sample results for an UDP traceroute and an nmap traceroute are seen in Figure 4.5 and 4.6. The RTT values for these probes remained approximately constant regardless of the TTL value of the probe.

```
traceroute to 192.168.253.9 (192.168.253.9), 30 hops max, 60 byte packets
 1  192.168.159.130  0.117 ms  0.688 ms  0.329 ms
 2  192.168.1.1  61.903 ms  64.566 ms  55.946 ms
 3  192.168.1.10  69.765 ms  69.563 ms  78.546 ms
 4  192.168.1.15  64.996 ms  65.220 ms  65.167 ms
 5  192.168.1.20  71.978 ms  67.472 ms  60.800 ms
 6  192.168.253.9  61.190 ms  64.469 ms  57.560 ms
```

Figure 4.5: An UDP traceroute to a fake host in physical machine setup where DeTracer does not introduce any artificial delay. Each probe's RTT is approximately 65ms, regardless of the TTL value of the probe.

```
TRACEROUTE (using proto 1/icmp)
HOP RTT      ADDRESS
1   0.87 ms   192.168.159.130
2   55.87 ms 192.168.1.1
3   63.05 ms  192.168.1.10
4   62.82 ms   192.168.1.15
5   77.70 ms 192.168.1.20
6   71.11 ms 192.168.253.9
```

Figure 4.6: An nmap traceroute to a fake host in physical machine setup where DeTracer does not introduce any artificial delay. The RTT values remain constant at approximately 65ms.

Additionally, the faster processing allowed us to introduce artificial delays based upon the incoming probe's TTL value. We implemented artificial delay by putting the Scapy program to sleep for a random amount of time. Figure 4.7 shows the code snippet in DeTracer that caused the random delay. The Scapy process added the TTL of the incoming packet to a random number between 0 and 1, then divided that number by 64. The process would then sleep for that many seconds. The time spent sleeping was specified by a floating point number and therefore the process could sleep for a fractional number of seconds. The number 64 was derived from experimentation to create feasible RTT values for each hop in the traceroute path but did not represent a thoroughly researched value. This number was also tunable to the environment in which DeTracer was running. The amount of variance in the time spent sleeping increased for each hop in the traceroute path. The amount of artificial delay induced in the first hop was between 0ms and 15ms, 0ms and 31 ms for the second hop, and 0ms and 46ms for the third hop. As the TTLs of the traceroute probes increased, the artificial delay increased because it was being drawn from a distribution that included larger numbers.

```
time.sleep((pkt.ttl + random.random())/64)
```

Figure 4.7: Code snipped that implements delay in DeTracer.

An example result for a traceroute probe can be seen in Figure 4.8. The RTT values for this traceroute were closer to what we would expect to see for a true topology. The RTT for different probes returning from the same hop count had close to the same value with some variance built in while the RTT was higher for hops further away in the trace. The RTT values did not have to be strictly increasing as the hop count increases because it was possible for probes sent with higher TTL values to have smaller RTT values than ones with lower TTL values. However, the RTT values should have trended upwards as the hop count increased, which was the exact behavior we saw in Figure 4.8.

```
traceroute to 192.168.253.9 (192.168.253.9), 30 hops max, 60 byte packets
 1  192.168.159.130  0.982 ms  0.913 ms  0.441 ms
 2  192.168.1.1  91.677 ms  91.053 ms  88.432 ms
 3  192.168.1.10  174.400 ms  205.109 ms  186.951 ms
 4  192.168.1.15  217.053 ms  204.755 ms  204.776 ms
 5  192.168.1.20  228.868 ms  203.897 ms  217.606 ms
 6  192.168.253.9  223.198 ms  307.111 ms  202.472 ms
```

Figure 4.8: A traceroute to a fake host behind DeTracer, while running on a physical machine.

The addition of artificial delay combined with the increased computing resources solved the decreasing RTT problem that was encountered in the VM setup. Figure 4.9 shows the RTT values of an nmap traceroute increased as the hop count increased. In Figure 4.8 and 4.9, the first hop had an RTT value of less than 1ms because that was the response time to the host running DeTracer, which was not part of the actual deceptive topology. Every hop after the first hop represented a faked host part of the false network topology.

```
TRACEROUTE (using proto 1/icmp)
HOP RTT       ADDRESS
1   0.56 ms   192.168.159.130
2   102.14 ms 192.168.1.1
3   196.20 ms  192.168.1.10
4   299.11 ms   192.168.1.15
5   264.68 ms 192.168.1.20
6   315.41 ms 192.168.253.9
```

Figure 4.9: An nmap traceroute to a fake host behind DeTracer, while running on a physical machine.

Having to add additional processing power in order to handle even one user running a traceroute has some important implications for DeTracer and its deployment feasibility. We posit that DeTracer's intensive processing requirements leaves the deception highly vulnerable to even modest Denial of Service (DOS) attacks. It is likely that an attacker could reveal the deception simply by specifying that more probes be sent simultaneously using the -N traceroute option [15]. An attacker could run a traceroute that only sent a few probes simultaneously and see normal RTT values, then run a traceroute that sent large numbers of probes simultaneously, perhaps from multiple machines at the same time.

From our experimentation, the traceroute with a small number of simultaneous probes would display reasonable RTT values while the traceroute with a large number of simultaneous probes would return abnormal RTT values, where the delay became unreasonably high as the number of packets being handled by DeTracer increased. This behavior was identical to that depicted in Figure 4.2 but only appeared when the DeTracer host received a large number of traceroute probes simultaneously. We leave the further investigation and resolution of this DOS pitfall to future work.

4.1.3 Public IP Setup

The final phase of our experimentation was to deploy DeTracer on the public Internet. To successfully deploy DeTracer, we required the upstream router and network to permit spoofed-source IP addresses. In typical operation, DeTracer assumed that it was operating at the network ingress, where it observed all packets destined to the entire network prefix. However, in our deployment, DeTracer ran on a machine with a single globally routable IP address where only packets to that single destination arrived at the host. We therefore adjusted DeTracer to operate at a single IP address and present a deceptive path to any traceroute probe directed at that single IP address. Modifying DeTracer for this use case was a matter of simplification. Instead of having to check the destination IP address of the incoming traceroute probe and determine whether that host should be active, DeTracer simply had to inspect the TTL of the incoming packet. Then, DeTracer would respond with a spoofed ICMP message using the source IP address corresponding to the hop at which the incoming probe would have expired in the fake topology. If the TTL of the incoming traceroute probe was high enough to reach barf.cmand.org (38.68.239.50) in the false topology, DeTracer responded with an ICMP message with 38.68.239.50 as the source IP address. We hope to deploy DeTracer at an ingress router to provide deception over a larger network in the future.

In this setup, depicted in Figure 4.10, DeTracer ran on a Fedora 19 VM. The physical machine the VM ran on had much more computing resources than the physical machine in Section 4.1.1. In addition, the host was operating as a web server, which remained accessible throughout our experimentation with DeTracer. We found that DeTracer was able to deceive every method of traceroute that we tested it against. In our experimentation, we noted that when DeTracer was overloaded with packets, it exhibited the same unrealistic delay problems that we found in Section 4.1.1. It appeared that when more than approximately 16 simultaneous probes were in flight, that the RTT values returned by DeTracer began to grow exponentially and lose the realistic delay properties we saw in Section 4.1.2. When we used the CAIDA Ark infrastructure to probe the deceptive host, the deceptive topology was presented but the RTT values seen were unpredictable. Some traces exhibited the increasing delay with some variance that we would expect to see in a real trace but others had unrealistically large RTTs (higher than 5 seconds for example). An example of a trace with realistic delay is depicted in Figure 4.11.

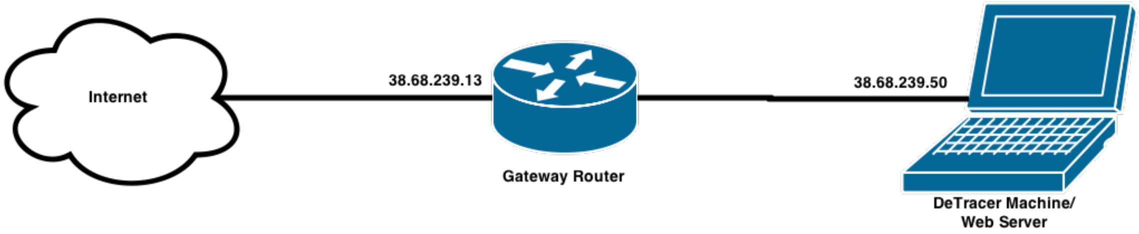


Figure 4.10: The logical layout of the DeTracer host in the Public IP setup.

```
traceroute from 46.20.241.26 to 38.68.239.50
  1.1:  46.20.241.25    87.114 ms
  2.1:  46.20.251.33    0.695 ms
...
 10.1:  38.127.193.146    88.014 ms
 11.1:  38.68.238.13    88.634 ms
 12.1:  82.128.128.21    141.727 ms
 13.1:  124.235.67.33    176.315 ms
 14.2:  103.7.146.140    165.579 ms
 15.1:  23.92.63.80    194.143 ms
 16.2:  131.173.201.39    201.595 ms
 17.1:  178.104.206.188    231.021 ms
 18.1:  115.4.245.211    238.672 ms
 19.1:  18.187.58.76    253.040 ms
 20.1:  *
 21.2:  1.162.93.232    571.000 ms
 22.2:  99.51.213.43    691.460 ms
 23.1:  222.96.12.66    482.984 ms
 24.1:  35.97.41.110    503.285 ms
 25.1:  213.110.125.46    889.228 ms
 26.1:  17.97.111.250    360.243 ms
 27.1:  178.55.74.161    372.773 ms
 28.1:  115.192.134.11    393.628 ms
 29.1:  *
 30.1:  58.125.71.50    461.610 ms
 31.1:  4.93.82.132    714.448 ms
 32.1:  105.39.221.28    760.313 ms
 33.2:  108.227.107.71    475.934 ms
 34.2:  131.160.37.215    963.697 ms
 35.2:  146.148.157.227    499.265 ms
 36.1:  181.250.205.174    516.664 ms
 37.2:  182.13.127.230    580.447 ms
 38.1:  93.155.237.183    951.322 ms
 39.1:  5.155.163.98    565.988 ms
 40.1:  38.68.239.50    558.967 ms
```

Figure 4.11: A traceroute from an Ark node to DeTracer on the public Internet depicting realistic RTT values.

An example of a traceroute from an Ark node to DeTracer that returned infeasible RTT values is seen in Figure 4.12. As DeTracer was flooded with packets, the RTT in the probe increased all the way up to 3105ms for hop 30 in the traceroute. As the rate of incoming packets decreased toward the end of the traceroute in Figure 4.12, the delay values returned to feasible values. Another interesting result of this phase of experimentation was that it disproved our theory in Section 4.1.1 as to the cause of the unrealistic delay problem. We originally postulated that the unrealistic delays were a result of the DeTracer host's CPU being overloaded. However, in this experiment, the DeTracer host's CPU usage never went above 10% even when it was overloaded with packets and returning poor RTT results. Our use of the sleep function in DeTracer blocked the program from handling a traceroute packet quickly and moving to the next packet in the queue. Without the time.sleep() function, DeTracer was able to handle up to 60 simultaneous traceroute probes while returning constant RTT values. This is demonstrated by the traceroute results in Figure 4.13.

```
traceroute from 46.20.241.26 to 38.68.239.50
  1.1:  46.20.241.25    0.581 ms
  2.1:  46.20.251.33    5.303 ms
...
  9.1:  154.54.28.109    88.529 ms
 10.1:  38.127.193.146    87.937 ms
 11.1:  38.68.238.13    88.689 ms
 12.2:  174.122.170.68    217.962 ms
 13.1:  193.129.47.125    495.234 ms
 14.1:  203.155.133.64    578.824 ms
 15.1:  *
 16.2:  122.195.214.177    1117.945 ms
 17.1:  *
 18.1:  175.7.72.230    1561.162 ms
 19.1:  39.144.111.154    1790.308 ms
 20.3:  16.66.16.198    2088.839 ms
 21.2:  121.215.26.151    2674.231 ms
 22.1:  201.200.241.128    2951.338 ms
 23.1:  86.46.194.229    2546.826 ms
 24.3:  99.2.30.31    1796.244 ms
 25.1:  106.19.154.35    3222.026 ms
 26.1:  211.198.137.20    2856.547 ms
 27.2:  72.22.243.91    1967.971 ms
 28.1:  89.113.21.77    2401.777 ms
 29.3:  85.188.150.218    2802.994 ms
 30.1:  91.231.165.233    3405.960 ms
 31.1:  96.21.210.229    2813.394 ms
 32.2:  172.210.71.45    2514.100 ms
 33.1:  199.211.68.190    3105.480 ms
 34.2:  39.143.109.63    1792.432 ms
 35.1:  37.193.101.203    2157.786 ms
 36.1:  24.124.130.74    1521.912 ms
 37.1:  71.228.6.187    1342.317 ms
 38.2:  210.183.59.228    550.271 ms
 39.1:  144.110.180.108    590.116 ms
 40.1:  38.68.239.50    546.006 ms
```

Figure 4.12: A traceroute from an Ark node to DeTracer on the public Internet depicting unrealistic RTT values.

```
traceroute from 196.216.3.6 to 38.68.239.50
  1.1:  196.216.3.2    1.353 ms
  2.1:  196.216.3.130    0.438 ms
...
 11.1:  154.54.6.169    252.486 ms
 12.1:  38.127.193.146    252.320 ms
 13.1:  38.68.238.13    263.369 ms
 14.1:  54.204.190.94    276.256 ms
 15.1:  1.129.222.254    283.534 ms
 16.3:  216.86.64.202    284.812 ms
 17.1:  92.93.50.114    286.145 ms
 18.2:  108.193.152.208    286.336 ms
 19.1:  99.194.80.171    273.403 ms
 20.1:  221.84.123.120    278.096 ms
 21.1:  68.39.204.165    275.005 ms
 22.1:  189.47.216.207    277.652 ms
 23.1:  104.90.154.223    276.043 ms
 24.1:  109.25.247.4    282.534 ms
 25.2:  205.100.192.196    278.901 ms
 26.1:  181.154.197.8    276.327 ms
 27.2:  216.134.187.53    279.523 ms
 28.3:  81.169.153.204    279.647 ms
 29.3:  143.235.86.11    289.382 ms
 30.2:  126.100.233.29    282.836 ms
 31.2:  133.121.130.56    277.917 ms
 32.1:  110.133.90.239    277.174 ms
 33.1:  15.120.29.16    275.732 ms
 34.2:  36.79.105.13    277.750 ms
 35.2:  129.49.145.132    285.069 ms
 36.1:  37.180.110.203    283.349 ms
 37.3:  200.39.23.57    279.990 ms
 38.2:  128.131.61.199    278.806 ms
 39.1:  208.161.64.71    280.431 ms
 40.1:  147.149.174.144    282.803 ms
 41.1:  54.74.15.233    280.214 ms
 42.1:  38.68.239.50    254.734 ms
```

Figure 4.13: A traceroute from an Ark node to DeTracer on the public Internet with a single queue, no artificial delay added, and numerous simultaneous traceroutes being performed.

This led us to attempt to perform load balancing using multiple nfqueue queues and multiple DeTracer instances. We created 32 nfqueue queues and 32 separate DeTracer instances, each attached to its own queue of packets. While this approach was ineffective in our original VM setup, it vastly improved DeTracer's performance when running on a machine with enough processing power to support numerous DeTracer instances. When running 32 instances of DeTracer, we were able to return realistic RTT values for as many as 100 simultaneous probes. We did not do further stress testing to determine the maximum load that this iteration of DeTracer could handle.

In this phase of experimentation, we presented several false topologies to traceroute probes. The first topology introduced 29 false nodes between the prober and the end of the traceroute path, each of which was a randomly selected IP address. As a result, each time a traceroute was performed to barf.cmand.org, DeTracer returned a completely different path to the end host. Responding with a path of random IP addresses was not the only manner in which DeTracer was deployed, but merely the first topology we tested in this experiment. Figures 4.11 and 4.12 depict sample results of the topology presented. The deception began at hop 12 in both Figure 4.11 and Figure 4.12. More traceroute results ran from various Ark nodes are contained in the appendix.

The second false topology we presented emulated a network path performing load-balancing between three separate paths to an end host. Figure 4.14 depicts the three possible paths to the end host. Load-balancing decisions were based upon the header values of source IP address, destination IP address, packet protocol, source port, and destination port. Packets for which those header values were the same were directed to the same path. This was accomplished by computing the hash of the 5-tuple of header values, and choosing the path presented based upon that hashed value. Real routers perform load-balancing based on the hash value of the same 5-tuple of header values. As such, DeTracer mimicked real-world router load-balancing.

```
traceroute to 38.68.239.50 (38.68.239.50), 30 hops max, 60 byte packets
  1  216.66.30.101 (216.66.30.101)    11.194 ms  1.271 ms    0.335 ms
  2  62.115.49.173 (62.115.49.173)    2.218 ms   0.217 ms    0.430 ms
  3  213.248.85.106 (213.248.85.106)    0.841 ms  0.254 ms  26.671 ms
  4  154.54.31.9 (154.54.31.9)   1.160 ms     0.415 ms  35.634 ms
  5  154.54.80.162 (154.54.80.162)    7.415 ms   0.734 ms   47.750 ms
  6  154.54.6.169 (154.54.6.169)    7.074 ms    9.660 ms    191.300 ms
  7  38.127.193.146 (38.127.193.146)    6.926 ms  12.823 ms 190.396 ms
  8  38.68.238.13 (38.68.238.13)    7.888 ms    19.142 ms   207.268 ms
  9  38.68.213.2 (38.68.213.2)   65.649 ms
                  (38.68.213.1) 19.799 ms
                  (38.68.213.5)  227.760 ms
 10  38.68.215.128 (38.68.215.128)   76.154 ms
                    (38.68.215.129)  20.692 ms
                    (38.68.215.130) 238.638 ms
 11  38.68.215.5 (38.68.215.5)    101.808 ms
                  (38.68.215.1)  138.944 ms
                  (38.68.215.3 ) 255.797 ms
 12  38.68.239.13 (38.68.239.13)    176.163 ms
                   (38.68.239.1) 201.011 ms
                   (38.68.239.1) 256.995 ms
 13  38.68.239.1 (38.68.239.1)    278.909 ms   293.682 ms
                  (38.68.239.103)  543.203 ms
 14  38.68.239.101 (38.68.239.101)    382.917 ms  392.524 ms
                    (38.68.239.117)  453.987 ms
 15  38.68.239.50 (38.68.239.50)    444.500 ms  397.532 ms  358.433 ms
```

Figure 4.14: An UDP traceroute to DeTracer on the public Internet portraying a load-balancing network.

The third topology we presented was a simple routing loop. In this topology, DeTracer returned a traceroute path that traversed the same three IP addresses twice before finally arriving at the end host. An example result from this experiment is shown in Figure 4.15.

```
Tracing route to barf.cmand.org [38.68.239.50]
over a maximum of 80 hops:
  1     1 ms     1 ms     1 ms  192.168.1.1
  2     8 ms     9 ms     9 ms  67.188.20.1
  3     9 ms    10 ms    10 ms  68.87.198.33
  4    14 ms    21 ms    13 ms  68.87.192.45
  5    22 ms    18 ms    15 ms  68.86.86.102
  6    15 ms    16 ms    16 ms  68.86.86.126
  7    99 ms    30 ms    18 ms  50.248.118.238
  8    20 ms    15 ms    16 ms  154.54.7.173
  9    59 ms    59 ms    59 ms  154.54.30.54
 10    70 ms    64 ms    65 ms  154.54.6.86
 11    72 ms    76 ms    73 ms  154.54.44.86
 12    93 ms    89 ms    94 ms  154.54.29.222
 13    86 ms    93 ms    89 ms  154.54.41.53
 14    86 ms    86 ms    86 ms  38.127.193.146
 15    87 ms    95 ms    89 ms  38.68.238.13
 16   160 ms   149 ms   146 ms  38.68.239.100
 17   159 ms   170 ms   151 ms  38.68.239.110
 18   167 ms   182 ms   169 ms  38.68.239.100
 19   188 ms   200 ms   199 ms  38.68.239.110
 20   212 ms   226 ms   205 ms  38.68.239.100
 21   218 ms   230 ms   219 ms  38.68.239.110
 22   205 ms   338 ms   204 ms  38.68.239.50
```

Figure 4.15: A sample traceroute to DeTracer on the public Internet portraying a routing loop.

The final topology we presented was designed to demonstrate the possibility of spoofed AS adjacencies and its importance. In this experiment, we chose the path depicted in Figure 4.16. The false results began at hop 16 in the diagram. This false traceroute path began at 64.69.48.1 and 64.69.59.3. These IP addresses belonged to a prefix advertised by AS 15147, which was the U.S. Department of Homeland Security (DHS). The next IP address, 101.16.13.1, was advertised by China Unicom. 175.45.176.55 belonged to a prefix owned by Star Joint Venture Co., the only ISP in North Korea. 112.91.128.17 also belonged to China Unicom. We selected this route because North Korea depended on China Unicom as its main AS provider [26]. Additionally, we wanted to demonstrate that it was possible for DeTracer to present false AS adjacencies that would be extremely alarming.

```
Tracing route to barf.cmand.org [38.68.239.50]
over a maximum of 80 hops:
  1     1 ms     2 ms     2 ms  192.168.1.1
  2    20 ms    10 ms    11 ms  67.188.20.1
  3    20 ms     9 ms    10 ms  68.87.198.33
  4    17 ms    15 ms    15 ms  68.86.143.10
  5    17 ms    33 ms    14 ms  68.86.86.102
  6    18 ms    16 ms    16 ms  68.86.86.126
  7    24 ms    17 ms    15 ms  50.248.118.238
  8    16 ms    23 ms    21 ms  154.54.7.173
  9    61 ms    65 ms    60 ms  154.54.30.54
 10    65 ms    65 ms    69 ms  154.54.6.86
 11    80 ms    74 ms    73 ms  154.54.44.86
 12    86 ms    92 ms    89 ms  154.54.29.222
 13    87 ms    94 ms    87 ms  154.54.41.53
 14    90 ms    87 ms    87 ms  38.127.193.146
 15    88 ms    91 ms    87 ms  38.68.238.13
 17   161 ms   168 ms   169 ms  64.69.48.1
 18   198 ms   184 ms   188 ms  64.69.59.3
 19   197 ms   183 ms   202 ms  101.16.13.1
 20   213 ms   207 ms   196 ms  175.45.176.55
 21   213 ms   213 ms   225 ms  112.91.128.17
 22   229 ms   235 ms   234 ms  38.68.239.1
 23   228 ms   227 ms   221 ms  38.68.239.50
```

Figure 4.16: A sample traceroute to DeTracer on the public Internet portraying the Department of Homeland Security being AS adjacent to China Unicom and STAR-KP in North Korea.

CHAPTER 5: CONCLUSION

This thesis delved into an active defense strategy to combat network mapping attacks. We demonstrated an effective method of presenting false network topologies to all variants of traceroute probes without disrupting normal network operations such as serving web pages. This deceptive technique, dubbed DeTracer, can be implemented on an ingress router to protect an entire network or on a single node to emulate the existence of an arbitrarily large fake topology. We next examine questions generated by our research that are left for future research.

5.1 Future Work

While our implementation accomplishes the essentials of presenting a deceptive network topology, there remain numerous points of interest that warrant further investigation and improvement. Future work includes incorporating more realistic delay models into the deception, adding support for Internet Protocol version 6 (IPv6), handling non-traceroute inbound traffic, and measuring the success of the deception.

5.1.1 Investigate Realistic Delay Models

One key aspect of future work in DeTracer is pinpointing the underlying cause of instances of unrealistic delay values presented to an attacker. These unrealistic delays are especially acute when DeTracer is overwhelmed with packets. Some possible explanations for the poor deceptive performance in these situations are the additional delay incurred by running inside a VM instead of directly on hardware and simply the delay of bringing a packet to user space before performing the packet parsing and crafting the spoofed response. Additionally, DeTracer's use of the Python sleep function in order to inject artificial delay into responses to traceroute probes may be detrimental to DeTracer's performance. While this problem seemed to be remedied by employing multiple queues and DeTracer processes, it may be possible to inject delay in another manner that does not cause a bottleneck for traceroute probe responses. It is possible that if we were able to shift some of the load to kernel space, we would see better performance from DeTracer in high usage situations.

After fixing causes of excessive queuing in DeTracer, we can delve into creating more realistic delay values for traceroute probes. In our implementation, we used the incoming probe's TTL value with some randomness added in order to present an increasing, non-deterministic delay to the attacker. There are several possible methods of implementing a more realistic delay model worth investigating:

- One possibility is to use empirical data of traceroute probes from the University of California San Diego's CAIDA research project to build a model for our network [39].
- Another option is to have a separate node in our network with a real, physical link that we can ping or probe in order to obtain a realistic round trip time for a packet inside our network. The delay that our deception returns would be a function of this round trip time and the number of hops the traceroute probe was expected to take before timing out. One potential disadvantage with this method is that it is unlikely for there to be much fluctuation in the delay times or in packet loss on a link that is reserved specifically for this purpose. There is also the additional work of maintaining a server simply in order to respond to pings. Additionally, this deployment scheme would make DeTracer dependent upon a second machine, which could potentially fail.
- We could ping nodes within our network to get this information, but this comes with the problem of generating unwanted and unnecessary traffic on our network. This method keeps our deception entirely contained within our network and practices good Internet citizenship.
- A fourth possible solution would be to ping a web server outside of our network and take advantage of the inherent load and queuing of the greater Internet to give us the realistic delay and loss that we need.

5.1.2 IPv6

Our implementation of DeTracer currently operates exclusively in Internet Protocol version 4 (IPv4). While this is a good starting point, the increasing adoption of IPv6 means that we will have to implement support for IPv6 as well [40]. There is nothing fundamentally different about the way traceroute functions in IPv6 versus IPv4. Thus, we believe the addition of IPv6 support to DeTracer would be accomplished simply by adding an IPv6 filter to ingress traffic and changing the IP addresses in the Python graph object that DeTracer ingests to IPv6 addresses.

One dynamic that IPv6 adds is the sheer vastness of the address space. The relative sparsity of real hosts in such a large address space may make implementing a credible deception scheme more straightforward than in the IPv4 space. In the IPv4 address space, it is feasible for an attacker to perform traceroutes for every host on a particular target subnet. This same exhaustive approach becomes impractical in the IPv6 space because of the large number of possible IP addresses on each network. This might mean that it is less likely for an attacker to send traceroute probes to a target network, making our deception scheme less valuable. However, because host discovery is such a difficult task in IPv6, traceroutes to known active hosts may prove to be an effective means of finding live hosts on a network. This would make our deception scheme more valuable against attackers because it would allow us to present fake nodes to an attacker and rendering them incapable of finding the true hosts on our network.

5.1.3 Investigate Effect on Normal Network Traffic

Before DeTracer can be feasibly deployed in a production network, we must fully understand its effect on the legitimate traffic on the network. If we cannot be sure that DeTracer will not have a negative impact of the intended operations of the network (whether it be web browsing, email, or any other application), we cannot hope to deploy it in a live environment. We investigated this problem to a small extent by placing a web server behind the DeTracer deception in our experiments. While the web server functionality was unaffected, this small experiment is far from sufficient proof that DeTracer will not have a detrimental effect on normal network operations. We would like to investigate whether DeTracer has a significant effect on network bandwidth, latency for other applications running on the network, and its impact on the workload for ingress routers where it is deployed.

5.1.4 Handling Non-traceroute Traffic

In order to present a truly convincing false topology, DeTracer will have to incorporate responses to other types of network traffic besides traceroute probes. For example, if a node appears in a traceroute presented by DeTracer but does not correspond to a real host on the network, DeTracer should then respond to all traffic sent to that host. If an attacker were to ping one of these faked hosts in DeTracer's current implementation, they would receive no response and likely suspect that there is something suspicious about the traceroute results they have seen. There is likely no legitimate reason why a host would respond to an ICMP traceroute probe but not respond to a direct ICMP Echo Request. Firewalls that block ICMP messages or hosts that do not respond to ICMP messages typically filter all ICMP messages, not just certain types. It is conceivable that a firewall could block ICMP Echo Reply messages but not ICMP Time Exceeded messages, but this is extremely unlikely. Hence, DeTracer must be able to detect traffic that is headed toward fake hosts in its deceptive topology and send spoofed responses for all traffic, or at least ICMP Echo Requests. Otherwise, an attacker can easily discover that there is no actual host at the reported IP address. An alternative solution to this problem would be to combine DeTracer with a tool such as LaBrea Tarpit that would respond to traffic on the network destined for non-existent hosts.

We foresee two possible deployment strategies for DeTracer. One complication that DeTracer must deal with in order to present a believable deception is what to do in the case that a real host in our network is running a publicly addressable service such as a web server or mail server. Suppose for example that we have a real web server on our network that is two hops away from the ingress router on which DeTracer is running. Suppose further that in DeTracer's presented fake topology, the web server is ten hops away from the ingress router. When an attacker runs a traceroute to the web server, they will see RTT values that correspond to the web server being ten hops away, specifically the delay will be longer. These increasing RTT values are caused by DeTracer injecting artificial delay into the traceroute responses as the TTL increases. This behavior makes the TTLs more realistic. If they simply connect to the web page, however, the RTT of the TCP packets coming from the web server will reflect that the web server is only two hops away from the ingress router, and the RTT of these packets will be much lower than those seen in the traceroute probes. These conflicting RTT values would be very suspicious to an attacker and likely reveal that there is deception happening on the defended network. A convincing fake topology in a setup like this would either have to delay types of traffic besides traceroute traffic leaving the network, or present a fake topology that closely mirrored the true topology in terms of the number of hops between the ingress router and the destination host on the network.

The second and more simplistic use case is where DeTracer is deployed on a network where there are no legitimate hosts and no services running on the network. In this scenario, the deception presented by DeTracer would be more difficult to detect because there would be less information leaked about the state of the network behind DeTracer. An attacker would not be able to perform the attack outlined in the previous example in order to decloak the deception because all traffic returned from the network would be originating from DeTracer.

5.1.5 Hardware and Performance Requirements

All of our experiments involved simple setups where we used the equipment we had readily available. Future work should investigate specific hardware requirements that DeTracer needs for production deployment without impacting other network applications. If we determine the amount of computing resources DeTracer needs in order to process packets at a certain rate, we can deploy DeTracer in a manner that would prevent it from being inundated with packets and performing poorly. This line of inquiry goes along with the future work outlined in Section 5.1.1 in that it would allow us to determine how much traceroute traffic DeTracer could feasibly handle without presenting unrealistic delay times. Although employing load-balancing and multiple DeTracer instances greatly improved DeTracer's ability to handle an influx of traceroute probes, we would like to optimize DeTracer so that it is not so susceptible to being overloaded with traceroute probes. We hope to address this weakness by investigating what are the hardware requirements of DeTracer to keep up with an influx of traceroute traffic, and by making DeTracer operate more efficiently. Some possible methods of improving DeTracer's efficiency include moving as much of the deception work into kernel space as possible (instead of bringing the packets up to user space) and implementing DeTracer using a programming language that allows for superior performance, such as C.

In this research, we focused on deployed DeTracer on a Linux host that was functioning as a router. In the future, we would like to deploy the same DeTracer functionality directly on routers. One particularly interesting research area would be the use of Software-Defined Networking (SDN) to present false responses to traceroute probes [41]. However, OpenFlow currently does not support flow rules based on the TTL field of incoming packets, which prevents us from being able to detect incoming traceroute probes. We believe that once this TTL filtering is added, it will be possible to implement DeTracer functionality in SDN switches and routers.

5.1.6 Multiple Ingress Points

Most production networks do not have only one ingress router to allow traffic in and out of their network. As such, DeTracer needs to support the capability to be deployed at several ingress points to a network and have each router present a false topology that is consistent with the others. While we did not attempt to deploy DeTracer at multiple points in this work, we believe that it would be a straightforward extension of the current functionality. Because DeTracer simply ingests a Python graph object and which node it is running on in that graph, we can create multiple DeTracer instances on different nodes simply by passing in the same graph object to all of them and altering which node that DeTracer is configured with on a per node basis.

5.1.7 Deceptive Topologies Presented

In this thesis, we presented two fake topologies on the public Internet. There are numerous further tricks that we might play with our deceptive topologies. We discussed some of these in depth in Section 2.4. We only began to delve into all the possible false topologies that DeTracer has the capability of presenting. We leave to future work the deployment of false topologies that are more interesting, tailored to a defender's specific goal, or designed to influence an adversary in a particular manner.

5.1.8 Success of Deception

Perhaps the most essential future work for this research is a study into the believability of the presented deception. There are two levels of believability that we seek. The first level involves deceiving an automated topology collection engine such as Ark. We believe that this level of believability is easier to obtain and that DeTracer in its current form is fully capable of deceiving an automated topology measurement tool.

The second level of believability involves deceiving a human operator who is probing the defended network. This level of believability should prove more difficult to obtain. In our work, we did not perform any experiments pitting DeTracer against a human adversary. In order to examine DeTracer's believability to a human adversary, we would like to perform a

Red Team analysis of our DeTracer tool. A useful experiment would be to have Red Team members attempt to map several experimental networks. Some of these networks would be real topologies and some would be false topologies presented by DeTracer. The Red Team, armed with the knowledge that some of the presented networks were not real, would probe all the networks in an attempt to determine which ones were real and which were faked. This would give us an idea of how effective DeTracer is in practice. Additionally, it will likely reveal shortcomings to our deception model not presently accounted for. A thorough Red Team analysis would tell us what other information that DeTracer is leaking that indicates a fake topology. For example, the Red Team might observe bandwidth limitations on the target network, RTT values that are unrealistic, or some other weakness that we have not foreseen. These clues might reveal to a potential attacker that there is deception on the defended network. The revelation of these weaknesses and information leaks would allow us to improve DeTracer.

5.2 Concluding Remark

Every day, we are bombarded with reminders of the importance of the cyber domain to our lives and to the DOD. It is integral that we continually strive to create novel technologies for securing our military and civilian networks. This thesis has presented a novel implementation of active network defense through topology deception. While DeTracer remains a proof of concept in its infancy, we believe that it can complement other network defense strategies.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Traceroute Experiment Results

A.1 Long Path Traceroute Results

A.1.1 Ark Commands Run

```
11 bre-de trace 38.68.239.50
12 sjc2-us trace 38.68.239.50
13 pry-za trace 38.68.239.50
14 jfk-us trace 38.68.239.50
15 gva-ch trace 38.68.239.50
16 dac-bd trace 38.68.239.50
17 cbg-uk trace 38.68.239.50
18 ams3-nl trace 38.68.239.50
19 bjc-us trace 38.68.239.50
20 yyz-ca trace 38.68.239.50
21 per-au trace 38.68.239.50
```

A.1.2 Ark Results

```
traceroute from 192.168.1.130 to 38.68.239.50
  1.1:  192.168.1.1    0.582 ms
  2.1:  *
  3.1:  83.169.158.102    8.571 ms
  4.1:  88.134.193.137    9.369 ms
  5.1:  88.134.238.165    15.524 ms
  6.1:  88.134.202.19    17.748 ms
  7.1:  62.115.9.9    13.162 ms
  8.1:  213.155.135.92    16.131 ms
  9.1:  62.115.142.15    21.940 ms
 10.1:  130.117.14.89    17.083 ms
 11.1:  130.117.48.114    23.747 ms
 12.1:  154.54.62.77    28.558 ms
 13.1:  154.54.28.109    107.783 ms
 14.1:  38.127.193.146    110.316 ms
 15.1:  38.68.238.13    116.908 ms
 16.1:  27.97.169.70    158.175 ms
 17.1:  116.185.154.172    180.054 ms
 18.1:  132.151.96.66    195.040 ms
 19.1:  17.44.248.173    205.984 ms
 20.1:  120.14.251.208    308.950 ms
 21.1:  63.157.7.107    355.214 ms
 22.1:  166.107.142.229    555.238 ms
 23.1:  21.223.246.128    598.470 ms
 24.1:  108.133.166.116    396.015 ms
 25.1:  168.230.9.28    413.122 ms
 26.1:  59.180.28.180    559.295 ms
 27.1:  111.171.169.106    423.691 ms
 28.1:  71.146.122.233    435.879 ms
 29.2:  163.231.59.187    430.234 ms
 30.1:  44.236.209.213    501.057 ms
 31.1:  67.122.52.80    542.323 ms
 32.1:  91.207.199.59    541.049 ms
 33.1:  37.122.213.33    828.111 ms
 34.1:  123.3.17.130    625.317 ms
 35.1:  20.155.146.152    1056.007 ms
 36.1:  136.217.179.9    607.045 ms
 37.2:  41.169.163.86    493.022 ms
 38.1:  21.4.139.61    732.351 ms
 39.1:  58.149.50.55    527.595 ms
 40.1:  195.54.119.26    541.505 ms
 41.1:  52.85.173.110    737.368 ms
 42.1:  176.204.38.32    559.884 ms
 43.2:  114.184.173.93    914.421 ms
 44.1:  38.68.239.50    574.913 ms


traceroute from 196.216.3.6 to 38.68.239.50
  1.1:  196.216.3.2    1.890 ms
  2.1:  196.216.3.130    0.450 ms
 3.1:  196.37.155.178    0.438 ms
 4.1:  168.209.1.162    4.119 ms
 5.1:  168.209.246.65    174.730 ms
 6.1:  149.6.148.129    164.477 ms
 7.1:  130.117.49.89    164.913 ms
 8.1:  130.117.50.202    175.216 ms
 9.1:  154.54.30.185    244.583 ms
10.1:  154.54.80.162    243.254 ms
11.1:  154.54.6.169    252.910 ms
12.1:  38.127.193.146    252.671 ms
13.1:  38.68.238.13    251.206 ms
14.1:  215.47.166.220    303.079 ms
15.2:  59.9.245.222    323.556 ms
16.1:  104.85.118.189    419.160 ms
17.1:  1.85.221.83    344.495 ms
18.2:  138.145.155.197    407.774 ms
19.2:  172.232.8.170    510.902 ms
20.1:  36.133.214.225    394.457 ms
21.1:  141.165.77.157    428.702 ms
22.2:  39.143.46.143    428.350 ms
23.1:  199.79.109.185    481.618 ms
24.1:  192.2.57.182    516.589 ms
25.1:  223.137.123.195    540.096 ms
26.1:  217.153.44.47    812.123 ms
27.1:  208.174.28.50    603.000 ms
28.2:  163.29.135.65    515.511 ms
29.1:  156.79.249.127    753.645 ms
30.1:  70.163.124.31    775.834 ms
31.1:  *
32.1:  204.96.2.59    570.862 ms
33.1:  162.108.236.74    602.353 ms
34.1:  210.222.193.156    625.951 ms
35.1:  142.244.141.83    698.822 ms
36.1:  208.104.240.166    1099.121 ms
```

```
 37.2:  126.86.97.46    1159.302 ms
 38.1:  146.124.116.46    1221.549 ms
 39.2:  203.192.253.161    788.855 ms
 40.1:  122.239.196.100    899.257 ms
 41.1:  27.44.225.69    931.592 ms
 42.1:  38.68.239.50    1317.835 ms


traceroute from 216.66.30.102 to 38.68.239.50
  1.1:  216.66.30.101    6.797 ms
  2.1:  62.115.49.173    2.053 ms
  3.1:  213.248.85.106    2.027 ms
  4.1:  154.54.31.9    0.998 ms
  5.1:  154.54.80.162    7.011 ms
  6.1:  154.54.6.169    7.035 ms
  7.1:  38.127.193.146    7.035 ms
  8.1:  38.68.238.13    7.806 ms
  9.2:  130.204.142.138    67.400 ms
 10.1:  198.167.199.73    88.386 ms
 11.1:  83.106.89.112    103.985 ms
 12.1:  147.198.199.2    127.334 ms
 13.1:  67.134.16.105    126.744 ms
 14.1:  46.220.40.138    213.006 ms
 15.1:  142.114.32.93    234.708 ms
 16.2:  139.82.1.163    168.245 ms
 17.2:  197.29.234.111    250.909 ms
 18.1:  166.179.111.6    337.334 ms
 19.1:  86.91.34.196    373.704 ms
 20.1:  *
 21.1:  150.183.233.211    244.667 ms
 22.1:  137.169.185.17    274.667 ms
 23.3:  189.67.52.59    475.935 ms
 24.1:  70.234.111.14    647.209 ms
 25.1:  15.27.16.41    317.138 ms
 26.2:  187.245.114.3    326.049 ms
 27.2:  81.97.174.234    637.042 ms
 28.2:  91.129.248.73    917.164 ms
 29.2:  86.74.138.87    371.072 ms
 30.1:  146.185.98.239    408.599 ms
 31.1:  134.240.150.31    879.030 ms
 32.1:  68.248.22.112    913.226 ms
 33.1:  108.200.45.138    944.088 ms
 34.1:  90.180.212.119    1306.285 ms
 35.3:  132.251.95.5    604.637 ms
 36.1:  94.118.243.82    945.297 ms
 37.1:  38.68.239.50    953.701 ms
```

```
traceroute from 46.20.241.26 to 38.68.239.50
  1.1:  46.20.241.25    87.114 ms
  2.1:  46.20.251.33    0.695 ms
  3.1:  46.20.254.82    0.664 ms
  4.1:  46.20.252.34    0.601 ms
  5.1:  213.242.73.29    89.288 ms
  6.1:  4.69.168.8    27.612 ms
  7.1:  130.117.14.93    12.044 ms
  8.1:  154.54.73.241    11.185 ms
  9.1:  154.54.28.109    88.113 ms
 10.1:  38.127.193.146    88.014 ms
 11.1:  38.68.238.13    88.634 ms
 12.1:  82.128.128.21    141.727 ms
 13.1:  124.235.67.33    176.315 ms
 14.2:  103.7.146.140    165.579 ms
 15.1:  23.92.63.80    194.143 ms
 16.2:  131.173.201.39    201.595 ms
 17.1:  178.104.206.188    231.021 ms
 18.1:  115.4.245.211    238.672 ms
 19.1:  18.187.58.76    253.040 ms
 20.1:  *
 21.2:  1.162.93.232    571.000 ms
```

```
22.2:  99.51.213.43    691.460 ms
23.1:  222.96.12.66    482.984 ms
24.1:  35.97.41.110    503.285 ms
25.1:  213.110.125.46    889.228 ms
26.1:  17.97.111.250    360.243 ms
27.1:  178.55.74.161    372.773 ms
28.1:  115.192.134.11    393.628 ms
29.1:  *
30.1:  58.125.71.50    461.610 ms
31.1:  4.93.82.132    714.448 ms
32.1:  105.39.221.28    760.313 ms
33.2:  108.227.107.71    475.934 ms
34.2:  131.160.37.215    963.697 ms
35.2:  146.148.157.227    499.265 ms
36.1:  181.250.205.174    516.664 ms
37.2:  182.13.127.230    580.447 ms
38.1:  93.155.237.183    951.322 ms
39.1:  5.155.163.98    565.988 ms
40.1:  38.68.239.50    558.967 ms


traceroute from 64.71.191.54 to 38.68.239.50
 1.1:  64.71.191.53    0.239 ms
 2.1:  213.248.67.105    0.202 ms
 3.1:  62.115.143.122    0.842 ms
 4.1:  62.115.34.74    1.451 ms
 5.1:  154.54.7.173    3.082 ms
 6.1:  154.54.30.54    50.186 ms
 7.1:  154.54.6.86    73.678 ms
 8.1:  154.54.44.86    73.824 ms
 9.1:  154.54.29.222    74.396 ms
10.1:  154.54.41.53    74.694 ms
11.1:  38.127.193.146    74.241 ms
12.1:  38.68.238.13    75.137 ms
13.2:  40.190.42.222    226.476 ms
```

```
14.1:  201.91.84.182    229.968 ms
15.1:  97.194.104.247    248.457 ms
16.2:  160.83.168.225    188.072 ms
17.1:  131.166.75.249    321.496 ms
18.1:  1.184.191.118    207.762 ms
19.2:  134.3.93.151    473.877 ms
20.1:  94.242.81.93    594.608 ms
21.1:  108.213.232.106    371.232 ms
22.3:  177.47.2.186    702.256 ms
23.2:  84.96.194.16    409.038 ms
24.2:  78.128.148.173    703.569 ms
25.1:  17.43.196.228    494.395 ms
26.2:  106.248.93.78    322.492 ms
27.1:  39.10.157.108    616.318 ms
28.1:  99.232.225.20    353.290 ms
29.2:  177.37.1.243    695.695 ms
30.2:  49.175.154.114    392.766 ms
31.1:  203.85.165.178    708.203 ms
32.2:  8.75.124.167    421.154 ms
33.3:  122.121.1.122    437.394 ms
34.1:  39.179.162.122    452.205 ms
35.2:  132.190.93.39    474.284 ms
36.1:  191.143.111.91    487.436 ms
37.1:  *
38.1:  88.28.25.162    516.072 ms
39.1:  161.159.88.76    535.279 ms
40.1:  112.90.54.136    546.031 ms
41.1:  38.68.239.50    542.862 ms
```

```
traceroute from 206.108.0.41 to 38.68.239.50
 1.1:  206.108.0.33    2.745 ms
 2.1:  206.108.0.10    0.266 ms
 3.1:  206.108.0.15    0.249 ms
 4.1:  204.232.76.73    0.408 ms
 5.1:  38.88.240.65    0.744 ms
 6.1:  66.28.4.197    9.663 ms
 7.1:  154.54.3.93    12.726 ms
 8.1:  154.54.40.73    19.092 ms
 9.1:  154.54.5.233    19.939 ms
10.1:  38.127.193.146    20.316 ms
11.1:  38.68.238.13    20.897 ms
12.2:  182.27.39.116    74.018 ms
13.1:  101.43.102.200    86.889 ms
14.1:  49.184.233.82    102.293 ms
15.2:  163.8.224.148    125.072 ms
16.1:  109.26.163.31    136.622 ms
17.1:  143.220.84.171    150.215 ms
18.1:  115.52.158.217    170.531 ms
19.1:  185.8.175.241    176.878 ms
20.1:  54.153.210.99    190.841 ms
21.1:  135.165.160.228    206.473 ms
22.1:  119.7.224.204    385.356 ms
23.1:  138.138.61.41    400.560 ms
24.1:  29.181.93.37    447.273 ms
25.2:  69.97.26.151    284.028 ms
26.1:  135.70.187.252    562.744 ms
27.1:  105.3.112.169    622.559 ms
28.1:  110.118.132.95    660.158 ms
29.1:  122.9.179.130    826.970 ms
30.1:  118.184.4.193    867.782 ms
31.1:  223.163.86.87    1018.383 ms
32.1:  176.97.201.236    939.263 ms
33.1:  128.198.26.221    1048.570 ms
34.1:  121.253.127.66    729.834 ms
35.1:  129.218.101.65    786.639 ms
36.1:  34.234.214.55    563.634 ms
37.1:  110.254.68.71    745.586 ms
38.1:  140.164.204.229    796.142 ms
```

```
 39.1:  81.73.31.165    943.929 ms
 40.1:  38.68.239.50    986.518 ms
```

```
traceroute from 113.197.9.170 to 38.68.239.50
  1.1:  113.197.9.169    0.221 ms
  2.1:  202.158.198.62    0.428 ms
  3.1:  202.158.194.8    26.594 ms
  4.1:  202.158.194.18    35.630 ms
  5.1:  202.158.194.242    47.623 ms
  6.1:  113.197.15.56    47.738 ms
  7.1:  202.158.194.121    191.296 ms
  8.1:  207.231.240.8    190.665 ms
  9.1:  198.71.45.24    207.773 ms
 10.1:  198.71.45.18    227.385 ms
 11.1:  198.71.45.14    238.502 ms
 12.1:  198.71.45.57    256.244 ms
 13.1:  192.122.175.14    256.895 ms
 14.1:  38.68.238.1    256.383 ms
 15.2:  50.101.34.13    309.619 ms
 16.2:  1.162.252.213    322.462 ms
 17.2:  177.172.172.59    669.165 ms
 18.1:  143.36.218.138    761.829 ms
 19.2:  61.207.127.182    1137.242 ms
 20.1:  145.122.123.210    721.473 ms
 21.1:  50.119.231.116    778.697 ms
 22.1:  13.119.180.30    771.056 ms
 23.1:  100.243.94.7    976.406 ms
 24.1:  32.88.200.85    999.187 ms
 25.2:  182.216.161.152    463.503 ms
 26.3:  163.23.113.8    481.112 ms
 27.1:  87.60.182.78    1024.297 ms
 28.1:  66.228.106.56    604.913 ms
 29.1:  118.75.79.124    699.239 ms
 30.1:  78.185.233.121    690.978 ms
 31.1:  90.160.70.156    1014.179 ms
 32.1:  195.199.31.230    1034.665 ms
 33.1:  103.246.89.99    1464.301 ms
 34.1:  156.29.82.103    776.733 ms
 35.1:  105.204.171.97    813.046 ms
 36.1:  67.19.38.183    1115.123 ms
 37.1:  49.79.31.182    1154.460 ms
 38.1:  107.38.217.89    856.002 ms
 39.1:  139.137.92.92    690.867 ms
 40.1:  90.144.254.155    699.605 ms
 41.1:  1.223.202.197    710.223 ms
 42.1:  56.114.83.146    793.422 ms
 43.1:  38.68.239.50    1245.049 ms


traceroute from 119.40.82.245 to 38.68.239.50
  1.3:  119.40.82.241    0.308 ms
  2.1:  210.4.78.5    0.562 ms
  3.1:  210.4.78.225    1.317 ms
  4.1:  114.130.21.65    37.478 ms
  5.1:  114.130.1.5    38.930 ms
  6.1:  125.22.195.133    40.380 ms
  7.1:  182.79.245.81    244.631 ms
  8.1:  38.122.147.121    245.186 ms
  9.1:  154.54.0.238    246.565 ms
 10.1:  154.54.7.53    283.614 ms
 11.1:  154.54.29.117    284.523 ms
 12.1:  154.54.31.98    312.057 ms
 13.1:  154.54.41.205    317.192 ms
 14.1:  38.127.193.146    317.667 ms
 15.1:  38.68.238.13    317.529 ms
 16.2:  74.34.102.233    599.666 ms
 17.1:  58.118.73.117    807.790 ms
 18.1:  219.70.97.240    1053.057 ms
 19.1:  195.147.188.169    840.766 ms
 20.1:  33.248.88.81    1121.435 ms
 21.1:  220.137.226.44    902.882 ms
 22.2:  27.56.252.76    833.945 ms
 23.1:  56.242.109.105    968.768 ms
 24.1:  179.78.101.236    1014.310 ms
 25.2:  97.229.142.195    508.206 ms
 26.1:  35.99.43.62    822.865 ms
 27.1:  159.152.154.37    821.828 ms
 28.1:  149.209.157.174    733.310 ms
 29.1:  193.94.202.6    746.372 ms
 30.1:  146.133.151.90    594.404 ms
 31.1:  49.134.118.249    609.201 ms
 32.2:  189.162.77.211    643.271 ms
 33.1:  38.207.4.179    643.886 ms
 34.1:  213.21.102.146    986.213 ms
 35.1:  101.74.74.248    666.142 ms
 36.1:  78.195.207.104    683.414 ms
 37.1:  42.71.180.194    1001.907 ms
 38.1:  205.31.118.234    1006.739 ms
 39.3:  20.133.185.252    1374.153 ms
 40.1:  33.19.120.195    764.167 ms
 41.1:  159.44.29.14    794.256 ms
 42.1:  87.88.1.198    781.036 ms
 43.1:  211.149.105.143    1515.136 ms
 44.1:  38.68.239.50    939.820 ms


traceroute from 128.232.97.9 to 38.68.239.50
  1.1:  128.232.97.2    24.836 ms
  2.1:  193.60.89.5    0.341 ms
  3.1:  192.84.5.137    0.347 ms
  4.1:  192.84.5.94    0.414 ms
  5.1:  146.97.130.1    0.298 ms
  6.1:  146.97.37.185    2.623 ms
  7.1:  146.97.33.17    24.058 ms
 8.1:  62.40.124.197    5.931 ms
 9.1:  62.40.98.81    13.415 ms
10.1:  62.40.98.128    20.165 ms
11.1:  62.40.125.18    122.108 ms
12.1:  192.122.175.14    115.508 ms
13.1:  38.68.238.1    116.112 ms
14.1:  83.190.47.69    161.559 ms
15.2:  32.237.170.236    190.433 ms
16.1:  107.52.146.131    204.683 ms
17.1:  175.245.204.21    213.035 ms
18.1:  174.95.98.177    229.396 ms
19.2:  115.124.184.69    298.334 ms
20.1:  92.117.139.159    398.431 ms
21.1:  164.9.130.64    420.094 ms
22.2:  213.58.170.23    305.462 ms
23.1:  67.94.217.76    524.286 ms
24.1:  175.122.75.4    547.019 ms
25.1:  216.152.173.172    716.090 ms
26.1:  80.173.244.79    774.463 ms
27.1:  221.27.67.68    878.011 ms
28.1:  108.166.214.105    1009.787 ms
29.1:  187.122.160.239    915.597 ms
30.1:  41.214.136.66    1047.252 ms
31.1:  131.48.165.55    716.405 ms
32.2:  88.136.37.142    1116.349 ms
33.2:  217.135.162.75    516.134 ms
34.1:  *
35.1:  128.87.98.160    1062.742 ms
36.1:  31.69.163.252    757.621 ms
37.1:  54.187.115.67    794.010 ms
38.1:  108.107.92.70    825.927 ms
39.3:  76.29.37.161    570.037 ms
40.1:  145.99.44.31    1277.338 ms
41.1:  76.3.183.55    954.612 ms
```

```
 42.1:  38.68.239.50    581.911 ms
```

```
traceroute from 209.245.28.50 to 38.68.239.50
  1.1:  209.245.28.1    0.297 ms
  2.1:  209.245.29.209    213.777 ms
  3.1:  *
  4.1:  4.68.111.102    16.335 ms
  5.1:  154.54.6.53    16.538 ms
  6.1:  154.54.25.241    21.323 ms
  7.1:  154.54.29.117    35.718 ms
  8.1:  154.54.31.110    44.277 ms
  9.1:  154.54.5.233    44.382 ms
 10.1:  38.127.193.146    43.352 ms
 11.1:  38.68.238.13    44.228 ms
 12.1:  18.18.144.87    95.406 ms
 13.1:  82.56.215.53    112.773 ms
 14.1:  211.168.250.108    123.515 ms
 15.1:  189.192.74.67    199.977 ms
 16.1:  64.76.134.124    210.513 ms
 17.3:  214.74.61.225    725.089 ms
 18.2:  110.56.227.42    522.737 ms
 19.1:  70.95.225.170    368.393 ms
 20.1:  91.224.195.65    219.504 ms
 21.1:  69.138.196.219    231.022 ms
 22.1:  70.194.231.79    256.240 ms
 23.3:  208.16.126.169    268.154 ms
 24.1:  216.224.69.209    278.745 ms
 25.1:  73.77.177.234    301.560 ms
 26.1:  *
 27.1:  17.165.55.120    795.254 ms
 28.1:  *
 29.1:  134.79.187.184    364.811 ms
 30.1:  192.186.114.74    379.832 ms
 31.1:  74.150.253.153    401.617 ms
 32.1:  128.89.86.107    418.605 ms
 33.1:  73.47.203.79    422.792 ms
 34.2:  66.214.218.161    447.704 ms
 35.2:  221.191.119.55    463.524 ms
 36.2:  46.31.105.116    471.756 ms
 37.1:  8.248.49.93    499.201 ms
 38.1:  205.85.69.178    509.705 ms
 39.1:  210.211.101.58    528.034 ms
 40.1:  38.68.239.50    508.012 ms
```

```
traceroute from 193.0.0.168 to 38.68.239.50
  1.1:  *
  2.1:  193.0.3.4    1.502 ms
  3.1:  62.115.12.201    1.489 ms
  4.1:  213.155.136.252    1.657 ms
  5.1:  80.91.253.171    1.993 ms
  6.1:  80.239.160.14    4.419 ms
  7.1:  154.54.74.93    4.434 ms
  8.1:  154.54.58.69    85.291 ms
  9.1:  154.54.44.165    85.531 ms
 10.1:  154.54.30.41    85.440 ms
 11.1:  154.54.40.73    90.394 ms
 12.1:  154.54.5.233    91.705 ms
 13.1:  38.127.193.146    91.136 ms
 14.1:  38.68.238.13    96.465 ms
 15.1:  151.188.56.36    156.292 ms
 16.2:  144.200.130.126    912.120 ms
 17.1:  96.106.233.11    1042.972 ms
 18.2:  118.228.210.122    611.188 ms
 19.1:  151.18.251.246    963.487 ms
 20.1:  221.54.95.145    998.395 ms
 21.1:  210.76.194.4    905.889 ms
 22.1:  184.116.147.19    976.287 ms
 23.2:  214.142.131.137    399.440 ms
 24.1:  121.218.74.108    439.588 ms
 25.1:  149.238.135.19    309.418 ms
 26.1:  62.71.127.183    488.503 ms
 27.1:  55.126.250.69    781.324 ms
 28.2:  206.42.130.205    347.625 ms
 29.2:  71.51.88.164    798.272 ms
 30.3:  133.19.6.113    1349.853 ms
 31.1:  88.230.67.236    753.894 ms
 32.2:  45.20.72.119    839.857 ms
 33.1:  180.179.231.46    423.735 ms
 34.1:  178.169.59.197    454.456 ms
 35.1:  *
 36.1:  182.219.75.103    474.033 ms
 37.2:  126.27.117.23    509.846 ms
 38.1:  8.173.67.52    510.275 ms
 39.3:  15.172.160.115    528.064 ms
 40.1:  168.54.213.237    540.069 ms
 41.1:  36.232.27.248    566.347 ms
 42.1:  18.148.209.138    564.637 ms
 43.1:  38.68.239.50    560.039 ms
```

A.2 Load-Balancing Path Traceroute Results

A.2.1 Ark Commands Run

```
11 bre-de trace 38.68.239.50
12 sjc2-us trace 38.68.239.50
13 pry-za trace 38.68.239.50
14 jfk-us trace 38.68.239.50
15 gva-ch trace 38.68.239.50
16 dac-bd trace 38.68.239.50
17 cbg-uk trace 38.68.239.50
18 ams3-nl trace 38.68.239.50
19 bjc-us trace 38.68.239.50
```

```
20 yyz-ca trace 38.68.239.50
21 per-au trace 38.68.239.50
```

A.2.2 Ark Results

```
traceroute from 216.66.30.102 to 38.68.239.50
  1.1:  216.66.30.101    11.194 ms
  2.1:  62.115.49.173    2.218 ms
  3.1:  213.248.85.106    0.841 ms
  4.1:  154.54.31.9    1.160 ms
  5.1:  154.54.80.162    7.415 ms
  6.1:  154.54.6.169    7.074 ms
  7.1:  38.127.193.146    6.926 ms
  8.1:  38.68.238.13    7.888 ms
  9.1:  38.68.213.2    65.649 ms
 10.1:  38.68.215.128    76.154 ms
 11.1:  38.68.215.5    101.808 ms
 12.1:  38.68.239.13    176.163 ms
 13.1:  38.68.239.1    278.909 ms
 14.1:  38.68.239.101    382.917 ms
 15.1:  38.68.239.50    444.500 ms


 traceroute from 206.108.0.41 to 38.68.239.50
  1.1:  206.108.0.33    1.271 ms
  2.1:  206.108.0.10    0.217 ms
  3.1:  206.108.0.15    0.254 ms
  4.1:  204.232.76.73    0.415 ms
  5.1:  38.88.240.65    0.734 ms
  6.1:  66.28.4.197    9.660 ms
  7.1:  154.54.3.93    12.823 ms
  8.1:  154.54.40.73    19.142 ms
  9.1:  154.54.5.233    19.842 ms
 10.1:  38.127.193.146    19.799 ms
 11.1:  38.68.238.13    20.692 ms
```

```
12.1:  38.68.213.1    138.944 ms
13.1:  38.68.215.129    201.011 ms
14.1:  38.68.215.1    293.682 ms
15.1:  38.68.239.1    392.524 ms
16.1:  38.68.239.19    442.860 ms
17.1:  38.68.239.110    348.238 ms
18.1:  38.68.239.50    397.532 ms


traceroute from 64.71.191.54 to 38.68.239.50
 1.1:  64.71.191.53    0.229 ms
 2.1:  213.248.67.105    0.200 ms
 3.1:  62.115.143.122    0.828 ms
 4.1:  62.115.34.74    1.383 ms
 5.1:  154.54.7.173    2.422 ms
 6.1:  154.54.30.54    50.131 ms
 7.1:  154.54.6.86    72.883 ms
 8.1:  154.54.30.197    73.715 ms
 9.1:  154.54.6.169    73.890 ms
10.1:  38.127.193.146    74.760 ms
11.1:  38.68.238.13    75.550 ms
12.1:  38.68.213.1    288.600 ms
13.1:  38.68.215.129    365.540 ms
14.1:  38.68.215.1    430.721 ms
15.1:  38.68.239.1    466.943 ms
16.1:  38.68.239.19    397.358 ms
17.1:  38.68.239.110    543.799 ms
18.1:  38.68.239.50    463.674 ms


traceroute from 128.232.97.9 to 38.68.239.50
 1.1:  128.232.97.2    1.902 ms
 2.1:  193.60.89.5    0.436 ms
 3.1:  192.84.5.137    0.365 ms
 4.1:  192.84.5.94    0.449 ms
 5.1:  146.97.130.1    0.288 ms
```

```
  6.1:  146.97.37.185    2.656 ms
  7.1:  146.97.33.17    5.982 ms
  8.1:  62.40.124.197    5.947 ms
  9.1:  62.40.98.81    13.456 ms
 10.1:  62.40.98.128    20.190 ms
 11.1:  62.40.125.18    121.992 ms
 12.1:  192.122.175.14    115.480 ms
 13.1:  38.68.238.1    116.286 ms
 14.1:  38.68.213.2    429.693 ms
 15.1:  38.68.215.128    406.886 ms
 16.1:  38.68.215.5    454.594 ms
 17.1:  38.68.239.13    376.591 ms
 18.1:  38.68.239.1    527.622 ms
 19.1:  38.68.239.101    365.041 ms
 20.1:  38.68.239.50    465.528 ms


traceroute from 196.216.3.6 to 38.68.239.50
  1.1:  196.216.3.2    2.518 ms
  2.1:  196.216.3.130    0.495 ms
  3.1:  196.37.155.178    0.510 ms
  4.1:  168.209.1.162    2.718 ms
  5.1:  168.209.246.1    173.924 ms
  6.1:  149.6.148.129    164.583 ms
  7.1:  130.117.49.89    165.344 ms
  8.1:  130.117.50.202    174.463 ms
  9.1:  154.54.30.185    244.561 ms
 10.1:  154.54.80.162    244.990 ms
 11.1:  154.54.6.169    250.874 ms
 12.1:  38.127.193.146    252.487 ms
 13.1:  38.68.238.13    252.492 ms
 14.1:  38.68.213.1    592.185 ms
 15.1:  38.68.215.129    423.067 ms
 16.1:  38.68.215.1    460.109 ms
 17.1:  38.68.239.1    350.938 ms
 18.1:  38.68.239.19    366.278 ms
 19.1:  38.68.239.110    378.406 ms
 20.1:  38.68.239.50    386.945 ms
```

```
traceroute from 113.197.9.170 to 38.68.239.50
  1.1:  113.197.9.169    0.335 ms
  2.1:  202.158.198.62    0.430 ms
  3.1:  202.158.194.8    26.671 ms
  4.1:  202.158.194.18    35.634 ms
  5.1:  202.158.194.242    47.750 ms
  6.1:  113.197.15.56    47.746 ms
  7.1:  202.158.194.121    191.300 ms
  8.1:  207.231.240.8    190.396 ms
  9.1:  198.71.45.24    207.268 ms
 10.1:  198.71.45.18    227.760 ms
 11.1:  198.71.45.14    238.638 ms
 12.1:  198.71.45.57    255.797 ms
 13.1:  192.122.175.14    256.995 ms
 14.1:  38.68.238.1    256.367 ms
 15.1:  38.68.213.5    543.203 ms
 16.1:  38.68.215.130    581.553 ms
 17.1:  38.68.215.3    453.987 ms
 18.1:  38.68.239.103    358.433 ms
 19.1:  38.68.239.117    378.042 ms
 20.1:  38.68.239.199    388.167 ms
 21.1:  38.68.239.50    391.898 ms
```

```
traceroute from 46.20.241.26 to 38.68.239.50
  1.1:  46.20.241.25    0.862 ms
  2.1:  46.20.251.33    0.526 ms
  3.1:  46.20.254.82    0.894 ms
  4.1:  46.20.252.34    0.668 ms
  5.1:  213.242.73.29    0.868 ms
  6.2:  4.69.168.8    9.783 ms
  7.1:  130.117.14.93    10.537 ms
  8.1:  154.54.73.241    11.213 ms
  9.1:  154.54.28.109    88.475 ms
 10.1:  38.127.193.146    87.904 ms
 11.1:  38.68.238.13    89.649 ms
 12.1:  38.68.213.5    138.099 ms
 13.1:  38.68.215.130    162.051 ms
 14.1:  38.68.215.3    168.115 ms
 15.1:  38.68.239.103    193.852 ms
 16.1:  38.68.239.117    203.228 ms
 17.1:  38.68.239.199    216.219 ms
 18.1:  38.68.239.50    221.912 ms
```

```
traceroute from 193.0.0.168 to 38.68.239.50
  1.1:  *
  2.1:  193.0.3.4    1.416 ms
  3.1:  62.115.12.201    1.413 ms
  4.1:  213.155.136.252    1.736 ms
  5.1:  80.91.253.171    1.764 ms
  6.1:  80.239.160.14    4.305 ms
  7.1:  154.54.74.93    4.750 ms
  8.1:  154.54.39.110    82.441 ms
  9.1:  154.54.42.85    84.273 ms
 10.1:  154.54.40.73    89.193 ms
 11.1:  154.54.5.233    89.969 ms
 12.1:  38.127.193.146    89.547 ms
 13.1:  38.68.238.13    94.970 ms
 14.1:  38.68.213.1    152.239 ms
 15.1:  38.68.215.129    160.393 ms
 16.1:  38.68.215.1    181.663 ms
 17.1:  38.68.239.1    193.793 ms
 18.1:  38.68.239.19    210.008 ms
 19.1:  38.68.239.110    230.746 ms
 20.1:  38.68.239.50    228.402 ms
```

```
traceroute from 119.40.82.245 to 38.68.239.50
  1.1:  *
  2.1:  210.4.78.5    0.560 ms
  3.1:  210.4.78.225    1.067 ms
  4.1:  210.4.78.217    19.543 ms
  5.1:  103.7.248.61    1.552 ms
  6.1:  103.7.251.89    24.076 ms
  7.1:  180.87.37.57    68.241 ms
  8.1:  180.87.37.10    333.831 ms
  9.1:  80.231.130.5    324.446 ms
 10.1:  66.198.70.25    328.827 ms
 11.1:  216.6.57.2    323.126 ms
 12.1:  66.198.111.126    323.002 ms
 13.1:  154.54.12.17    322.807 ms
 14.1:  154.54.3.69    336.916 ms
 15.1:  154.54.31.125    329.646 ms
 16.1:  154.54.41.205    335.685 ms
 17.1:  38.127.193.146    329.426 ms
 18.1:  38.68.238.13    331.417 ms
 19.1:  38.68.213.5    400.419 ms
 20.1:  38.68.215.130    396.913 ms
 21.1:  38.68.215.3    438.247 ms
 22.1:  38.68.239.103    431.203 ms
 23.1:  38.68.239.117    518.172 ms
 24.1:  38.68.239.199    466.547 ms
 25.1:  38.68.239.50    516.260 ms
```

```
traceroute from 192.168.1.130 to 38.68.239.50
  1.1:  192.168.1.1    0.667 ms
  2.1:  *
  3.1:  83.169.158.102    7.879 ms
  4.1:  88.134.193.137    7.967 ms
  5.1:  88.134.238.165    13.139 ms
  6.1:  88.134.202.19    13.077 ms
  7.1:  62.115.9.9    22.554 ms
  8.1:  213.155.135.92    16.258 ms
  9.1:  62.115.142.15    18.958 ms
 10.1:  130.117.14.89    22.646 ms
 11.1:  130.117.48.114    21.741 ms
 12.1:  154.54.62.77    30.473 ms
 13.2:  154.54.28.109    107.404 ms
 14.1:  38.127.193.146    107.678 ms
 15.1:  38.68.238.13    112.403 ms
 16.1:  38.68.213.5    169.166 ms
 17.1:  38.68.215.130    371.587 ms
 18.1:  38.68.215.3    198.931 ms
 19.1:  38.68.239.103    302.206 ms
 20.1:  38.68.239.117    228.958 ms
 21.1:  38.68.239.199    306.732 ms
 22.1:  38.68.239.50    233.829 ms


traceroute from 209.245.28.50 to 38.68.239.50
  1.3:  209.245.28.1    0.194 ms
  2.1:  209.245.29.209    0.871 ms
  3.1:  *
  4.1:  4.68.111.102    16.028 ms
  5.1:  154.54.6.53    15.939 ms
  6.1:  154.54.25.241    21.691 ms
  7.1:  154.54.29.117    35.532 ms
  8.1:  154.54.31.110    44.292 ms
  9.1:  154.54.5.233    44.633 ms
 10.1:  38.127.193.146    43.707 ms
 11.1:  38.68.238.13    45.172 ms
```

```
12.1:  38.68.213.2    232.358 ms
13.1:  38.68.215.128    117.665 ms
14.1:  38.68.215.5    149.487 ms
15.1:  38.68.239.13    144.206 ms
16.1:  38.68.239.1    161.914 ms
17.1:  38.68.239.101    180.582 ms
18.1:  38.68.239.50    185.742 ms
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] Internet users. (n.d.). World Wide Web Consortium. [Online]. Available: http://www.internetlivestats.com/internet-users. Accessed Mar. 2, 2015.

[2] Department of Defense strategy for operating in cyberspace. (2011, Jul.). Department of Defense. [Online]. Available: http://www.defense.gov/news/d20110714cyber.pdf

[3] S. T. Trassare, "A technique for presenting a deceptive dynamic network topology," M.S. thesis, Dept. Comput. Sci., Naval Postgraduate School, Monterey, CA, 2013.

[4] E. J. Holdaway, "Active computer network defense: An assessment," DTIC, Maxwell Air Force Base, AL, Tech. Rep., April 2001.

[5] C. Trowbridge, "An overview of remote operating system fingerprinting," SANS Institute, Bethesda, MD, Tech. Rep., July 2003. [Online]. Available: http://www.sans.org/reading_room/whitepapers/testing/overview-remote-operating-system-fingerprinting_1231

[6] L. Spitzner, *Honeypots: Tracking Hackers*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] E. Dulaney and M. Harwood, *CompTIA Network+ N10-005 Authorized Exam Cram*. Indianapolis, IN: Que Publishing Co., December 2012.

[8] *Information Technology–Open Systems Interconnection–Basic Reference Model: The Basic Model*, ISO Standard 7498-7, 1994.

[9] J. Hawkinson and T. Bates, "Guidelines for Creation, Selection, and Registration of an Autonomous System (AS)," RFC 1930 (Best Current Practice), Internet Engineering Task Force, Mar. 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1930.txt

[10] Y. Rekhter *et al.*, "Border Gateway Protocol 4," RFC 4271, Internet Engineering Task Force, Jan. 2006. [Online]. Available: https://tools.ietf.org/html/rfc4271

[11] B. Huffaker *et al.*, "Macroscopic analyses of the infrastructure: Measurement and visualization of Internet connectivity and performance," in *Passive and Active Network Measurement Workshop (PAM)*, Amsterdam, The Netherlands, 2001.

[12] J. Postel, "Internet Protocol," RFC 791, Internet Engineering Task Force, Sep. 1981. [Online]. Available: https://www.ietf.org/rfc/rfc791.txt

[13] V. Cerf *et al.*, "Specification of Internet Transmission Control Program," RFC 675, Internet Engineering Task Force, Dec. 1974. [Online]. Available: https://tools.ietf.org/html/rfc675

[14] R. Braden, "Requirements for Internet Hosts," RFC 1122, Internet Engineering Task Force, Oct. 1989. [Online]. Available: https://tools.ietf.org/html/rfc1122

[15] V. Jacobsen, "traceroute (8) - Linux man page," 2001. [Online]. Available: http://linux.die.net/man/8/traceroute

[16] G. Lyon, "nmap reference guide (man page)," 2007. [Online]. Available: http://nmap.org/book/man.html

[17] M. Luckie *et al.*, "Traceroute probe method and forward IP path inference," in *Proc. 8th ACM SIGCOMM Conf. on Internet Measurement*, Vouliagmeni, Greece, 2008, pp. 311-324.

[18] R. Oppliger, "Internet security: Firewalls and beyond," *Commun. of the ACM*, vol. 40, no. 5, pp. 92–102, May 1997.

[19] Comodo Firewall. (n.d.). Comodo Group, Inc. [Online]. Available: http://www.comodo.com. Accessed Mar. 3, 2015.

[20] iptables. (n.d.). The Netfilter.org Project. [Online]. Available: http://www.netfilter.org. Accessed Mar. 3, 2015.

[21] D. Battista *et al.*, "Computing the types of the relationships between autonomous systems," in *INFOCOM 2003 Twenty-Second Annu. Joint Conf. of the IEEE Comput. and Commun.*, 2003, vol. 1, pp. 156-165.

[22] A. Toonk. (2010, Apr.). Chinese ISP hijacks the Internet. [Online]. Available: http://www.bgpmon.net/chinese-isp-hijacked-10-of-the-internet

[23] M. A. Brown. (2008, Feb.). Pakistan hijacks YouTube. [Online]. Available: http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1

[24] L. Lessig and R. W. McChesney. (2006, June). No tolls on the Internet. [Online]. Available: http://www.washingtonpost.com/wp-dyn/content/article/2006/06/07/AR2006060702108.html

[25] I. Paul. (2013, Mar.). The Pirate Bay admits to North Korean hosting hoax. [Online]. Available: http://www.pcworld.com/article/2030073/the-pirate-bay-admits-to-north-korean-hosting-hoax.html

[26] The Pirate Bay – North Korean hosting? No, it's fake. (P2). (2013, Mar. 5). [Online]. Available: https://rdns.im/the-pirate-bay-north-korean-hosting-no-its-fake-p2

[27] B. Munro. (2013, Mar.). North Korea hosting The Pirate Bay? Niet! [Online]. Available: http://beaglenetworks.net/post/44608108942/north-korea-hosting-the-pirate-bay-niet

[28] P. Mahadevan *et al.*, "The Internet AS-level topology: Three data sources and one definitive metric," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 1, pp. 17–26, 2006.

[29] K. Scarfone and P. Hoffman, "Guidelines on firewalls and firewall policy," *NIST Special Publication*, vol. 800, p. 41, 2009.

[30] B. Munro. (2013, Feb.). Star Wars traceroute. [Online]. Available: http://beaglenetworks.net/post/42707829171/star-wars-traceroute

[31] M. Zhang *et al.*, "How DNS misnaming distorts Internet topology mapping," in *USENIX Annu. Tech. Conf., General Track*, 2006, pp. 369–374.

[32] K. Keys, "Internet-scale IP alias resolution techniques," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 50–55, 2010.

[33] D. B. Berrueta. (2003). A practical approach for defeating nmap OS- Fingerprinting. [Online]. Available: http://nmap.org/misc/defeat-nmap-osdetect.html

[34] T. Liston. (2009). LaBrea. [Online]. Available: http://labrea.sourceforge.net/.

[35] A. Sebastian. (2009). Default Time To Live (TTL) values. [Online]. Available: http://www.binbert.com/blog/2009/12/default-time-to-live-ttl-values/

[36] P. Biondi. (2011). Scapy. [Online]. Available: http://www.secdev.org/projects/scapy

[37] P. Chifflier. (2012). Nfqueue-bindings. [Online]. Available: https://www.wzdftpd.net/redmine/projects/nfqueue-bindings/wiki

[38] E. Leblond. (2013, Jan.). Using nfqueue and libnetfilter_queue. [Online]. Available: https://home.regit.org/netfilter-en/using-nfqueue-and-libnetfilter_queue

[39] Center for Applied Internet Data Analysis. (n.d.). CAIDA. [Online]. Available: http://www.caida.org/home. Accessed Mar. 2, 2015.

[40] L. Colitti *et al.*, "Evaluating IPv6 adoption in the Internet," in *Passive and Active Network Measurement Workshop (PAM)*, 2010. [Online]. Available: http://www.pam2010.ethz.ch/papers/full-length/15.pdf

[41] J. J. Hughes, “Employing deceptive dynamic network topology through software-defined networking,” M.S. thesis, Dept. Comput. Sci., Naval Postgraduate School, Monterey, CA, 2014.

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California